ADST/WDL/TR--95-W003267C

Advanced Distributed
Simulation Technology

Software Architecture and Overview Document (SADOD) for ModSAF

Date: 28 April 1995
ModSAF Version 1.51

Prepared for:
STRICOM
U.S. Army Simulation, Training and Instrumentation Command
12350 Research Parkway
Orlando, FL 32826-3275

Contract N61339-91-D-001
Delivery Order ModSAF
Delivery Order 0058

LORAL
ADST Program Office
12151-A Research Parkway
Orlando, FL 32826

ADST/WDL/TR--95-W003267C

ADST

Advanced Distributed
Simulation Technology

Software Architecture and Overview Document (SADOD) for ModSAF

Date: 28 April 1995
ModSAF Version 1.51

Prepared for:
STRICOM
U.S. Army Simulation, Training and Instrumentation Command
12350 Research Parkway
Orlando, FL 32826-3275

Contract N61339-91-D-001
Delivery Order ModSAF
Delivery Order 0058

LORAL
ADST Program Office
12151-A Research Parkway
Orlando, FL 32826

REPORT DOCUMENTATION PAGE

Form approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED ModSAF Version 1.51

4. TITLE AND SUBTITLE Software Architecture and Overview Document (SADOD) for ModSAF	5. FUNDING NUMBERS Contract No. N61339-91-D-0001 Delivery Order ModSAF Delivery Order 0058
6. AUTHOR(S) Loral Advanced Distributed Simulation 50 Moulton Street Cambridge, MA 02138	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Loral System Company ADST Program Office 12151 -A Research Parkway Orlando, FL 32826	8. PERFORMING ORGANIZATION REPORT NUMBER ADST/WDL/TR-95-W003267C
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S Army Simulation, Training and Instrumentation Command (STRICOM) Naval Air Warfare Center, Training Systems Division 12350 Research Parkway Orlando, FL 32826-3275	10. SPONSORING ORGANIZATION REPORT NUMBER A005
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, distribution is unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT (Maximum 200 words)

This ModSAF 1.51 SADOD describes the ModSAF system architecture.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	17. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	17. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std Z39-18
298-102

ModSAF Software Architecture Design and Overview Document (SADOD) for ModSAF

14 April 1995

LADS Document Number 94070 v1.0

Short Contents

Table of Contents

1 Overview

The ModSAF *Software Architecture Design and Overview Document* or SADOD describes the ModSAF system architecture.

This chapter contains an introduction to Semi-Automated Forces (SAF) and ModSAF. This is followed by chapters on the ModSAF software architecture, the user interface architecture, the simulation architecture, and the command and control architecture. At the end of this document, a chapter is included describing *extend* or how to build on top of the ModSAF software, as well as some notes on memory and processing time.

There are many references to the ModSAF *Programmer's Reference Manual Index* (PRM) that point to other documents (or libraries) which contain detailed technical design and implementation information. The PRM Index is most easily accessed using the '`info`' hypertext documentation system.

An overview of the general concepts and some history behind Semi-Automated Forces followed by an overview of the ModSAF system are included.

1.1 ModSAF Q&A

Questions about ModSAF can be addressed to Loral Advanced Distributed Simulation through the ModSAF reflector.

Registering with the Reflector

To *register* with the reflector:

- Write to `listproc@public.tiig.ist.ucf.edu`. The text of the message must be `SUBSCRIBE MODSAF <your name>`. (Your e-mail address is taken from the e-mail header.)
- Send e-mail by writing to `modsaf@public.tiig.ist.ucf.edu`. Note: You must be registered to send mail to the reflector.

In addition, the reflector messages can be accessed on line via Gopher or World Wide Web.

If using Gopher, send to `gopher.tiig.ist.ucf.edu` and select Conferences, then MODSAF. If using a web browser to open the URL, send to `gopher://gopher.tiig.ist.ucf.edu:70/11/msis/conferences/modsafs`.

For questions about the reflector, write to `techspt@public.tiig.ist.ucf.edu`.

1.2 What is SAF?

Semi-Automated Forces (SAF) development began in the fall of 1987 at BBN under the Defense Advanced Research Projects Agency (DARPA) (Simulation Network) SIMNET project. It was designed using a concurrent engineering approach to accommodate evolving requirements to support combat and material development experiments in a Distributed Interactive Simulation (DIS) environment. It was quickly applied to support DIS military team training. Soldiers can now train in simulators on synthetic battlefields without the real world limitations of safety, cost, and available training areas and personnel. For example, the system was used to support large distributed exercises such as the WAREX '90 exercise that contained more than 800 manned and SAF vehicles. This system was also the primary technology used to support the DARPA sponsored 73 Easting Battle Reenactment that recreated a US/IRAQ engagement from the 1991 Persian Gulf war. This engagement was simulated with over 1000 SAF entities including ground vehicles, helicopters, dismounted infantry, and dynamic buildings (capable of being damaged or destroyed in the battle).

In order to meet these military training demands, the SAF system creates additional forces to populate the battlefield and can generate both opposing and supporting forces. Units composed of both manned and SAF vehicles can be created, as well. Many SAF vehicles can be controlled by a single operator and simulated through the use of one or more SAF computer systems. The system enables the operator to control all these vehicles by simulating the lower level decision making of units and vehicles, enabling the SAF operator to maintain supervisory control. The savings in operator and equipment costs produced by populating the battlefield with SAF rather than individual vehicle simulators makes large exercises feasible. SAF is also critical for DIS experimental studies where many independent trials are needed to provide statistically valid data. New subjects must be found and trained for each additional trial, making it essential to limit the number of subjects involved.

While the existing SAF system is able to support various applications, the potential of expanding the system by *extending* it with additional functionalities to support more applications is obvious. There are several reasons for the need of different organizations extending the SAF system:

- The spectrum of domain knowledge required to model all the battlefield operating systems is quite large. It ranges from the detailed modeling of the physical weapons systems, sensors, and environment to the modeling of human decision processes and military tactics.
- The range of implementation knowledge required to build a SAF system is also very broad, including such specialties as networking, user interfaces, computer graphics, distributed databases, analytic geometry, statistics, planning and software engineering.
- The requirements for many applications, while similar, vary enough that customizations of SAF will always be necessary. In fact, since weapons systems and tactics are continually changing, SAF will always continue to evolve.

In order to utilize other organizations' skills and resources and integrate their extensions into the SAF system, an open architecture system is needed. This architecture should allow independent groups to modify SAF for their individual applications, build further functionality on top of SAF, and still be able to trade and combine each others' modifications. Creating this general SAF system with a structural design sufficient to support a variety of research, training, and development projects is a challenge. This challenge is met by ModSAF.

1.3 What is ModSAF?

*Mod*ular Semi-Automated Forces (ModSAF) is an open software architecture system designed to fulfill the needs outlined in the above section. It provides uniform methodology and software support for creating and controlling entities on a simulated battlefield. ModSAF-simulated entities can behave autonomously; that is, they can move, fire, sense, communicate, and react without operator intervention. These entities can interact with each other as well as manned simulators, over a network supported by DIS.

An integral part of the ModSAF system is the Graphical User Interface (GUI). It automatically creates user-friendly interfaces with added functionality. The architecture provides the user with the ability to modify parameters at run-time. In addition, the user can easily create vehicles, simulate their behaviors, change their reactions, observe the battlefield, and monitor the vehicles' status.

A ModSAF entity is given extensive capabilities; for instance, it can drive over terrain avoiding obstacles, firing at enemy objects, using its sensors to detect radar, radioing messages to other entities, and executing missions. The goal of ModSAF is to replicate the outward behavior of simulated units and their component vehicle and weapon systems to a level of realism sufficient for training and combat development. ModSAF simulates an extensive list of entities. For Fixed Wing Aircraft (FWA) it simulates, for example, the F14D, the MIG29, the A10 and the SU25.

For Rotary Wing Aircraft (RWA) it can simulate the AH64, the OH58D, the Mi24, and the Mi28. For its ground forces, ModSAF simulates tanks (e.g. the M1 and the T72), Infantry Fighting Vehicles (IFV) (e.g. the M2 and the BMP), ADA (e.g. ZSU-23/4), and Dismounted Infantry (DI). Additional physical models also include Howitzers, Mortars, and Minefields as well as special effects models, such as smoke.

At the vehicle/weapons system level, ModSAF simulates entities by enabling them to execute a realistic range of basic actions inherent to the entity type. For example, a simulated tank can drive along a road, scan its turret, and turn in place. A simulated aircraft can ascend, orbit, and land. Similarly, weapons systems exhibit realistic rates of fire and realistic trajectories. ModSAF-simulated entities can also exhibit mobility, firepower, and catastrophic combat damage when hit by enemy fire. Their resources (both fuel and ammunition) are accurately depleted as they move and fire. Other simulated capabilities include intervisibility, target detection, target identification, target selection, fire planning, and collision detection. These capabilities are based on, but are not limited to, such appropriate factors as range, motion, activity, visibility, direction, orders, and evaluation of threat.

When a unit is simulated, ModSAF not only creates the ModSAF entities (such as a plane or tank) in a unit but also builds a structure corresponding to the unit hierarchy. Commands can then be issued to either the top-level units or to their subordinate units or vehicles. ModSAF interprets orders and generates the appropriate unit and vehicle behaviors and tactics.

The number of ModSAF entities that an operator can control is maximized by automating the low-level decision making of the entities. The efficiency of this automation is maximized by simulating only those features that are externally observable or significant to other simulation exercise participants. Therefore, low-level simulation is carried out by the SAF entities, although the operator still makes the high level, critical, tactical decisions for the forces being controlled, and has the ability to override or interrupt any automated behavior, at any time. *Tasks* govern the automated behavior performed by a ModSAF entity or unit. Examples of tasks include behavior such as move, orbit, avoid collisions, and search for enemy vehicles. ModSAF uses a set of representative tasks for both individual (vehicle) or collective (unit) tasks. These tasks are defined in terms of their characteristic parameters. ModSAF relies on standard military doctrine to supply default values for task parameters, however, the user is allowed to modify the default values in a variety of ways. A *task frame* consists of a set of related tasks that run at the same time. Task frames are typically composed of moving, targeting, and reacting tasks that all work together to accomplish the frames' goal. Furthermore, ModSAF vehicles and units can be commanded to execute a *mission* which is a collection of one or more task frames executed sequentially, with each task frame representing one *phase* of the mission. The user is allowed to specify the condition(s) that must be satisfied to permit transitioning between mission phases.

The ModSAF software architecture is flexible, hierarchical, and supportive. It is flexible so as to allow new behavioral representations be developed into ModSAF. In addition, its hierarchical structure makes it easy for the developer to add, delete, modify, or test a particular function. Finally, it provides user interface support for explanation, inspection, and modification of behavior.

2 Software Architecture

Organization of the various SAF system functionalities, from high-level end user inputs to low-level network communication, is critical. The first section describes the ModSAF Architecture. The functionalities embedded in the architecture are implemented with many software modules or libraries that can be replaced, modified, or extended as the need arises. These software modules are combined into application programs that make up the ModSAF system architecture. The application programs typically run on separate computers, distributed over a network. The most crucial element of distributed interactive simulation of SAF is the ability for these applications to communicate with each other. In particular, the SAF need to share their physical battlefield states and events among themselves as well as with any other applications on the network. Communication is handled through the DIS protocol. In addition, the SAF need to communicate command, control, and system administration information among themselves. This type of communication is handled through the Persistent Object (PO) protocol. Both of these protocols are described in more detail in Protocol Specifications. ModSAF maintains an open architecture allowing multiple developers to build on top of it. Most importantly, in order to be flexible and extensible, the software is written in a consistent, modular format. The objectives of ModSAF are supported by strict requirements, as outlined in the section entitled Design Methodology. Also described are the ModSAF library modules that implement the design objectives.

2.1 ModSAF Architecture

Supervisory Control

Supervisory control includes system administration functions such as placing units on the battlefield, creating missions for those units to execute, and monitoring or altering the execution of those missions during an exercise.

Supervisory control includes:

1. Graphical User Interface (GUI)
   The GUI provides mechanisms for an operator to influence the battle (also known as man-in-the-loop simulation). The operator can create, modify, delete vehicles as well as assign tasks (e.g. move, shoot, withdraw) to them. Overall battlefield events can be closely supervised on the Plan View Display (PVD).
2. Command and Control (C2) Simulations
   On an actual battlefield, the actions of ground/air vehicles are driven by the influence of a hierarchy of command and control nodes (C2 is the military term for the method by which orders are propagated through a military hierarchy). In ModSAF, these C2 nodes are simulated by a Task Manager, a collection of libraries

that provide the supervisory control needed to manage SAF units in an automated fashion.

Both methods provide supervisory control. In ModSAF, the emphasis is on facilitating human intervention.

Collective Behavior

An example of collective battlefield behavior is when a company of vehicles attacks an objective, thereby executing a sequence of actions as a group. Although these actions can be simulated by encoding the behaviors of the individuals, it is simpler to encode the group behavior.

Note: Constructive simulations generally stop at this level of simulation. Since the specific effects of groups are simulated using probablistic methods, there is no need to further subdivide the collective behavior.

Individual Behavior

The primary distinction between constructive simulations and SAF simulations is the encoding of individual entity behaviors. In ModSAF, entities are distinct, separable objects on the simulated battlefield. These include vehicles, dismounted infantry, missiles, command-and-control centers, etc. In order to accurately portray their outward behavior, the SAF system simulates the behavior of the humans and/or dynamic control processes which then guide the actions of these entities.

Physical Subsystems

In order to interact in the DIS battlefield, a SAF system must simulate not only the behaviors of entities but also the physical effects of those behaviors. This includes vehicle kinematics, weapons systems control, sensors, communications, etc. It is *only* at this level that all battlefield outcomes are resolved.

Support Services

One of the most fundamental services is rendered by a support library called LibPktValve. This library provides packet routing, distribution, transmission, filtering, and other network-related services to the separate modules in the higher layers of the ModSAF system within the same simulation exercise.

See section "Overview" in *LibPktValve Programmer's Reference Manual*. The transmission and receipt of packets to many independent modules is supported by LibQueue. Events sometimes need to be enqueued for later processing when it is inappropriate to act on the information right away. For example, when a collision packet is received off the network, the locally simulated vehicle of this packet is immediately notified, but the vehicle should not respond to the collision until its next tick.

See section "Overview" in *LibQueue Programmer's Reference Manual*.

To overcome math deficiences in the C programming language, a support library called libVecMat was created to provide a comprehensive suite of vector and matrix operations. Utilities for geometry-related functions and machine-independent random number generators are also available.

See section "Overview" in *LibVecMat Programmer's Reference Manual.*
See section "Overview" in *LibGeometry Programmer's Reference Manual.*
See section "Overview" in *LibRandom Programmer's Reference Manual.*

Overall, the ModSAF system provides a baseline of functionality upon which new systems can be developed. To this end, ModSAF implements separate, extensible modules that can be replaced, expanded, or augmented with additional modules as simulation technology advances.

2.2 ModSAF Application Programs

The ModSAF system architecture consists of software modules that are combined into application programs. These application programs communicate with one another via certain *protocols* (discussed in more detail in the next section).

SAFstation
The SAFstation or ModSAF's front-end workstation provides human supervisory control of ModSAF entities. Using the graphical user interface (GUI), the user can load scenarios, create battlefield entities, assign tasks to friendly/enemy vehicles, influence their behaviors, or simply examine the simulated battlefield on the Plan View Display (PVD). The user can take advantage of utilities such as the terrain tools, the drawing tools, the message log, the execution matrix, the artillery tool, etc., and also perform system administration such as freezing exercises, saving scenarios, creating overlays, changing user preferences, etc.

SAFsim The SAFsim or ModSAF's back-end simulator simulates collective and individual behaviors as well as the simulation of physical subsystems and, ultimately, battlefield interactions. The simulator maintains the state of the battlefield in an object-based database. This database contains *local* entities (updated by the SAFsim) and *remote* entities (updated by other SAFsims on the network). Therefore, any given SAFsim can update its own local entities, and transmit entity state packets (which are always the ground truth) to other simulators. It is the other simulators' responsibility to update the remote entities in their databases. This is the premise behind distributed simulation.

SAF Logger
ModSAF's Logger records and plays back exercises conducted on the simulated battlefield. The Logger records both the physical and behavioral states as well as ModSAF entity command and control hierarchies. The Logger also records packet usage. These

recordings enable the supervisor to restart scenarios from any point and to conduct a more detailed analysis of SAF behavior (After Action Review).

After Action Review

Stealth The Stealth (Preview or Flying Carpet) is a special application that allows the user to monitor battlefield events from any perspective. This is accomplished by attaching the stealth "vehicle" directly onto an entity (to view events from the entity's perspective) or via a "string" on the entity (to view events from a certain fixed distance from the entity's perspective). The ModSAF GUI also enables the stealth to be "teleported" to any entity being displayed on the PVD.

Manned Simulator

A manned simulator (vehicle simulator) is a human-operated simulator that resembles the interior of a tank or other platform. Manned simulators on the DIS network can detect and fire at SAF vehicles, and vice versa, as long as they are interacting on the same simulation exercise.

2.3 Protocol Specifications

In order for two or more simulation processes, on different computers, to communicate with each other, they must be able to transmit and receive information over the network.

The DIS protocol and the PO protocol enable the processes to communicate. These protocols are abstracted away from most parts of the ModSAF system, so that interactions take the form of database operations.

2.3.1 DIS Protocol

The Distributed Interactive Simulation (DIS) protocol creates a distributed simulation environment where battlefield entities simulated on different processors can interact. The basic concepts of DIS are an extension of the SIMNET program. (Note: The DIS and SIMNET protocols are different dialects but serve the same purpose. In the following discussions, references to DIS could be replaced with SIMNET in all cases.).

DIS is a set of protocols for linking dissimilar simulators together to form a common, consistent, shared, simulated world, thereby allowing these different types of simulators to interact in a team environment. The simulators can be distributed over a large geographical area where communications can be handled over local and wide area networks.

Applications currently used within DIS include manned simulators, dismounted infantry simulators, SAF simulators, command post simulators, and stealth. These applications can communicate with each other as long as they are running on the same DIS *exercise ID*. The exercise ID is a field in the DIS Protocol Data Unit (PDU) Header that identifies which simulation exercise the various applications are involved in.

SAF simulators act like manned simulators, in that they broadcast their states on the network. The types of information that DIS broadcasts for SAF vehicles includes entity states, weapons impacts, collisions, fire, radar, and weather conditions.

2.3.2 DIS Database Abstraction

The DIS environment is a constantly changing "database" of entities. Each vehicle, person, missile, and atmospheric or electromagnetic effect, is an object in this database.

In ModSAF, each object in the DIS database has a unique handle, called the `vehicle_id`. Note: not all DIS entities are vehicles; however, in ModSAF, "vehicle" is used to refer to each object.

Within a ModSAF process (SAFsim or SAFstation), state information regarding a given entity is attached to the object representing that entity in the global table. This is *ground truth* information: the most accurate information that is known about an entity at any instant in time. Information that is shared explicitly through the DIS protocol, such as position or vehicle type, will generally be nearly identical in all ModSAF processes. Information that is not shared in the DIS database, such as internal munition loads, is known only within the ModSAF process where that entity is being simulated.

When SAF vehicles interact with one another, they must restrict their queries to the information generally available (the state that is shared via the DIS protocol). However, interactions between modules within a simulation of a single vehicle can access any information about that vehicle made publically available by other modules within that process. By convention, a vehicle is always simulated entirely within a single process, so that there is no question whether a piece of state information will be available if requested.

In addition to transient state information, the DIS protocol communicates events (weapons fire, detonations, collisions, etc.). These events are not objects in the database of entities but, rather, influence those objects' state. For example, within the simulation of a vehicle is a module that monitors collisions. If a DIS packet that reports a collision with that vehicle is received off the network, the state of the vehicle's collision module changes to reflect that a collision has recently occurred. The collision module then transmits that information throughout the simulation process to all the other modules that are concerned with collisions.

Implementation

As mentioned, ModSAF applications are comprised of software modules or libraries, that define classes of objects. In the DIS environment, a simulated entity is represented by LibVTab that maintains the global table of DIS entities. LibClass facilitates hanging all the different information onto the database objects. LibEntity is the most frequently used DIS DB object subclass library, since it holds all the physical state information associated with each entity on the DIS network (location, type, velocity, etc.). Another library, libPBTab, provides a position-based vehicle table so modules can quickly find nearby vehicles (for sensor simulation, collision detection, etc.).

See section "Overview" in *LibVTab Programmer's Reference Manual.*
See section "Overview" in *LibClass Programmer's Reference Manual.*
See section "Overview" in *LibEntity Programmer's Reference Manual.*
See section "Overview" in *LibPBTab Programmer's Reference Manual.*

2.3.3 PO Protocol

Since much information about SAF behavior cannot be shared via the DIS Protocol, the Persistent Object (PO) protocol was created to provide a more flexible and scalable interface between the components of the SAF system.

The PO protocol enables the sharing of behavioral states, command and control information, and system administration. ModSAF software modules have unlimited access to all the information being broadcast via this protocol, provided they are running on the same PO *database ID.* The database ID is an identifier in the PO PDU Header that identifies which repository of persistent objects or "database" the applications interact with.

(As a bit of a historical note, the name persistent was derived from the notion that the objects being simulated should *persist* in spite of a computer system failure. In other words, if the computer responsible for maintaining the state of an object should fail, then responsibility for that object will *migrate* to another computer on the network.)

2.3.4 PO Database Abstraction

Like the DIS database, the PO database represents ground truth. It contains information about the unit organizations, exact tasks and missions assigned, the precise status of all simulated entities during the execution of those missions, the world state and control measures (e.g. point, line, area, route). Nearly exact copies of the information contained in the PO database can be found in each of the ModSAF processes in a given exercise. Using this abstraction allows the ModSAF processes to work together as one distributed system.

By interacting through the PO database, commands are distributed from any SAFstation to any SAFsim for execution. Load-leveling of vehicles between SAFsim processes is possible, allowing more optimal use of computing resources. This type of architecture supports systems with large numbers of SAFstations (e.g. for close supervisory control) or with large numbers of SAFsims (e.g. to generate large numbers of DIS entities with a small human staff). In addition, it allows objects from a failed SAFsim or SAFstation to be automatically taken over by a surviving system.

The PO protocol shares a large amount of information efficiently, provides real-time performance, and allows changes to the state of the objects. This is accomplished by periodically broadcasting Protocol Data Units (PDUs) which can describe an object, delete an object, request an object, etc. An implication of this concept of sharing information through a centralized PO database is that both local and remote changes appear identical to the ModSAF applications.

Implementation

All operations on the PO database are done through a library called libPO. This library contains the complicated semantics of the PO protocol that allow a consistent database to be maintained on all SAF processes. Another library, libClass, helps higher-level libraries to attach information to each object in the database.

See section "Overview" in *LibPO Programmer's Reference Manual.*
See section "Overview" in *LibClass Programmer's Reference Manual.*

2.4 Design Methodology

The ModSAF software architecture is an extensible set of software modules that allows rapid development and testing of new agents in the DIS simulated environment. New ideas for the command and control of automated DIS agents are implemented and tested without extensive re-development of already available SAF supporting code. In addition to being flexible and expandable, the ModSAF system must retain backward compatibility.

The ModSAF system has also successfully provided a testbed for the evaluation of intelligent control algorithms within a large scale real-time DIS simulation. The SAF/Soar project** is the first application of this testbed approach.

**Rosenbloom PS, Johnson WL, Schwamb KB, Tambe M, Jones RM, Koss F, Laird JE, Lehman JF, Rubinoff R. "Intelligent Automated Agents for Tactical Air Simulation: A Progress Report". Proceedings of the Fourth Conference on Computer Genertaed Forces and Behavioral Representation, Orlando, FL. May 4-6, 1994.

Opening the SAF system to the DIS community requires a new design methodology that:

- Allows replacement of individual subsystems without modification of the surrounding software by:
  - Layering the architecture
  - Supporting object-based programming
  - Defining rigorous interface specifications
  - Using data driven execution

- Allows the use of diverse hardware
- Allows subsystems to be written in almost any computer language
- Allows arbitrary distribution of subsystems across different hardware platforms at run-time
- Supports a time management scheme that allows for optimal scheduling and variable simulation

2.4.1 Replacement of Individual Subsystems

To easily replace individual subsystem modules, the ModSAF architecture uses four techniques: layering, object-based programming, rigorous interface specification, and data driven execution. The following sections describe these techniques in more detail.

2.4.1.1 Layering

Layering is the design methodology in which software modules are grouped into functional layers.

Programming is simpler and more manageable because software in one layer can only use functions and services available in lower layers. This also makes software easier to test, since the operation of each successive layer can be confirmed without having the entire system operational. Moreover, the use of layering in the ModSAF architecture allows subsystems to be replaced at various levels and facilitates reuse of software.

The following figure shows some of the layers involved in the representation of remotely simulated entities. This example of layering for remotely simulated entities is representative of the kind of layering used in all of ModSAF. Note: the placement of one module above another does not necessarily imply a dependence relationship; rather it ensures that the lower module does not depend on the higher module.

```
                Remote Vehicle Management (libremote)
----------------------------------------------------------------------
                    Entity State Cache (libentity)
----------------------------------------------------------------------
               Position-based Vehicle Table (libpbtab)
----------------------------------------------------------------------
            DIS Database Object Classification (libdisdbobj)
----------------------------------------------------------------------
                  Dead Reckoning Utilities (libdr)
----------------------------------------------------------------------
                     Math Utilities (libvecmat)
----------------------------------------------------------------------
                    Global Vehicle Table (libvtab)
----------------------------------------------------------------------
                     Packet Routing (libpktvalve)
----------------------------------------------------------------------
                           Operating system
```

Implementation

The layered nature of the ModSAF software gives rise to a number of implications in the way that software is written:

Service Model

Although some utility libraries are *stateless*, such as a math utility library that operates on arguments and returns a value, most utilities operate in a context which persists through time. These utilities, therefore, implement *classes* since they consist of both state and operations. When a utility library is written as a class it is generally referred to as a *service*. For example, the library that draws the two-dimensional map display provides a map drawing service. A higher layer user interface library creates an instance of a tactical map, then periodically calls functions to update the map's state.

Writing software as services not only "untangles" the software into logical layers, but also makes multiple instantiations natural. For example, a user interface with *two* independent tactical map displays could be created easily by calling the service model's create routine twice.

Polling vs Callback Events

Distribution of information through a layered system can be achieved in two ways: *polling* and *callbacks*. In a polling system, a higher layer module periodically asks a lower layer module for information. For instance, within the simulation of a vehicle, the module that detects collisions periodically asks the module that maintains entity state for the exact current location of the vehicle. This approach is appropriate when the state that is being queried changes frequently or constantly. However, polling is an inefficient approach when the state does not change frequently. Polling does not work when the item of interest is not a state change but instead a simulation *event*.

In the case of a simulation event, a *callback* mechanism is used. Specifically, the higher layer module provides the lower layer module with a function to call *if* the specified event occurs. For example, the module that monitors for collisions calls back to a higher layer module *if* a collision is detected.

Since many higher layer modules could be interested in any given event, a support library called libCallback has been written to help manage callback events.

See section "Overview" in *LibCallback Programmer's Reference Manual*.

2.4.1.2 Object-Based Programming

Object-Based programming cleanly separates the subsystems or library modules into classes of objects that are defined by a data structure and a family of functions that operate on that data structure. The data structures of the objects are separated into public and private components to minimize the interdependence between modules, and to simplify changing a module's implementation.

Traditional object oriented programming builds new objects from sub-objects using single inheritance. ModSAF employs object-based programming to perform this construction using *composition*, where larger classes are composed of subclasses. In ModSAF, there is a one-to-one correspondence between object classes and certain libraries. Note: the use of object-based techniques in the implementation of the modules is fully compatible with Ada techniques and ensures compatibility with Ada applications.

Object oriented programming as represented by CLOS, C++, and Objective C goes beyond object-based programming by including support from the programming language. However, the ModSAF architecture is implemented in Kernighan & Ritchie (K&R) C to maximize compatibility with a variety of hardware platforms, to provide maximum compatibility with other languages by avoiding run-time environment requirements, and to make maximum reuse of the large existing body of SAF C code. In addition, the use of C generally provides greater or equal run-time efficiency to that of other languages. Run-Time efficiency is important for SAF applications due to the continuing need to balance cost, number of vehicles being simulated, and modeling resolution.

Implementation

The ModSAF software is written using database abstractions for both the battlefield environment shared via the DIS protocol, and the command and control information shared via the PO protocol. Therefore, every application module that performs some part of the simulation, from physical subsystems up to supervisory control, is part of the ModSAF database.

An implication of distinguishing between the application and the database is the need for a mechanism to *compose* objects. Since each object in a database has only one handle, but may be composed of an arbitrary number of sub-parts, a way to multiplex that one handle into many is needed, so that the state information can be attached. A support library called libClass helps manage this task.

See section "Overview" in *LibClass Programmer's Reference Manual.*

In the ModSAF system, just as there are two database abstractions, there are correspondingly two *classes* of objects:

DIS DB Object Class

> Objects of the DIS DB (disdb) class correspond one-to-one with objects in the DIS database abstraction.

C2 Object Class

> Objects of the C2 (c2obj) class correspond one-to-one with objects in the PO database abstraction.

See section "Overview" in *LibDISDBObj Programmer's Reference Manual.*
See section "Overview" in *LibC2Obj Programmer's Reference Manual.*

An example of data driven execution is the definition of SAF vehicles. Items in the data files specify the parametric dynamics model to use (rotary wing aircraft, fixed wing aircraft, tracked ground vehicle, etc.), and the parameters of that model (maximum turn rate, fuel consumption, etc.). In addition, the network representation of the vehicle and dead reckoning thresholding parameters (See Section 4.8 [Dead Reckoning], page 32, for more info) are specified. Further, the weapons systems are specified by name, and vehicle-dependent weapons parameters are given for each (time to load the weapons, accuracy as a function of range, etc.). Since this data is specified for each vehicle, it is possible to change the performance characteristics of each vehicle without modifying any software (as long as the basic algorithms are capable of the desired behavior). Adding new kinds of vehicles often requires modification of data files only.

Another example of data driven execution is the software that draws top-down views of vehicles on the plan view display. A data file specifies the method for drawing various classes of vehicle types by choosing from a small set of graphic primitives (box, line, circle, etc.). Sizes, locations, and rotation (with the hull or turret) information is given for each attribute as well as the order in which to draw them.

Additionally, ModSAF allows the configuration of any entity to be modified *after* the vehicles have been created. In other words, parametric data can not only be modified by editing reader files, they can be modified during *run-time* through a "configure" button on the GUI. Though this is currently not a widely used practice, the architecture does support it.

Implementation

Although the simulation of SAF entities is spread over many software modules, the management of the parameters used by those modules is centralized in a single service module. Each subclass provides callbacks to this service module to convert the parameters in data files into an internal format. During dynamic composition of the object, the higher layer class (in this case the DIS DB object class) uses the parameter management service to determine which subclasses should be included, and what parameters should be passed to those subclasses (as described earlier, parameters can include system-level default values that specialize subclasses; for example, the object type of a vehicle is a parameter of its entity subclass).

Using this technique, the definition of which subclasses compose each object can reside entirely in a data file. Furthermore, using the data sharing services of the PO database, this dynamic composition has been extended to allow run-time reconfiguration of entities *after* they have been created.

See section "Overview" in *LibParMgr Programmer's Reference Manual.*

See section "Overview" in *LibParmEdit Programmer's Reference Manual.*
See section "Overview" in *LibComponents Programmer's Reference Manual.*

Since the ModSAF architecture is designed to be driven by data files, a central facility, called libReader, is responsible for reading and processing file-based data.

See section "Overview" in *LibReader Programmer's Reference Manual.*

There is also a service called libOTMatch that simplifies the construction of databases keyed by object types (object types are constants that represent entities, such as M1 Tank, Phoenix Missile, F14D Airplane, etc.).

See section "Overview" in *LibOtMatch Programmer's Reference Manual.*

2.4.4 Hardware Independence

Hardware independence enables ModSAF to run on different kinds of platforms (SGI, Sun, Mips, etc). The two main advantages are:

1. Already existing and available hardware can be used
2. The buy-in cost of hardware is minimized

To ensure hardware independence, the ModSAF system uses:

1. K&R C, because a K&R C compiler is almost always delivered with any Unix platform. Another benefit of using K&R C is that the developer does not have to pay for a compiler license, reducing the price of development workstations.
2. X windows and Motif for window and widget management on the GUI. Both X windows and Motif are supported on most workstations.

However, operating system independence cannot always be guaranteed since certain operations are not standardized (getting the value of the real time clock, sending and receiving raw ethernet packets, and other critical features). These differences are recognized through the use of conditional compilation. In practice, the ModSAF system does not use many operating system calls; therefore, there are few conditional sections in the code.

2.4.5 Programming Language Independence

It is a primary goal to allow the developer to use any language to build upon the ModSAF software. Using C libraries is very flexible because they can be called from virtually any other programming language, but the reverse is often not the case. Programming languages fall into two groups: (1) those that have a run-time environment (e.g., Ada, LISP, C++); (2) those that do not (e.g., C, Fortran, Assembly). In general, run-time environment languages can't be called by a main program that has no run-time environment or a dissimilar run-time environment. (Recent changes in Ada compilers are changing this however.) Therefore, if the developer wanted to install libraries in the system that used a run-time programming environment, it could be done by translating the main initializing program into the same language. Because almost all the functionality of the SAF system has been placed in the software libraries, the small main program body can be translated to any language very easily.

2.4.6 Distribution of Subsystems

The baseline ModSAF system assumes that the user interface process (SAFstation) and the simulation process (SAFsim) will be run on separate hardware platforms due to the real-time constraints of the simulation software and the stalling behavior of the X windows system. However, the software can be link-loaded into one executable for systems where hardware is scarce, and accuracy of simulation is not critical.

The sharing of entity appearance data is handled via the DIS protocol. However, there is a great deal of command and control data that cannot be shared using DIS. The sharing of this data is handled through the use of the Persistent Object protocol.

Implementation

The PO protocol defines classes of objects that are shared between hardware platforms. Procedures are defined to allow simultaneous editing of objects, and to ensure persistence of objects despite hardware failures. The protocol is implemented in the ModSAF architecture through libPO. LibPO provides an object-based database view of the persistent objects, as well as a paradigm for sharing information. Hardware independence of libPO is ensured such that when a change is made to the database, callback functions fire on all platforms, including the one which made the change. Software can, therefore, be written which does not distinguish between "local" and "remote" changes to the database. This allows the simulation of an object to be distributed across several platforms sharing state via the Persistent Object database.

See section “Overview” in *LibPO Programmer's Reference Manual.*

2.4.7 Time Management

Allocating computational resources to all the services and subclasses in a layered system requires a technique similar to callbacks. Most services that need computing resources need them periodically. For example, the module that reads the network often has no choice but to do so using periodic polling. For this reason, the ModSAF system uses a scheduler that optimizes the act of executing functions periodically. In addition, the scheduler can trigger functions that run once after a specified delay, when that approach is more efficient than the alternative of periodic polling.

In addition, the ModSAF system maintains 4 separate clocks:

1. Real-time millisecond clock
2. Internal simulation millisecond clock
3. Shared simulation millisecond clock
4. Wall clock

Implementation

The ModSAF scheduler is implemented as a service model. It provides the service of calling functions periodically (for "tick" based simulation) or after a delay. For example, the module that creates DIS DB class objects (libDISDBObj) requests that its *tick* routine (the customary name for a periodically invoked function) be executed no more than once every 67 milliseconds for each local DIS DB class object.

The distribution of computational resources to the subclasses which make up each locally simulated entity is performed by the DIS DB object class during a single tick routine. This allows simulation time to freeze during the tick, and allows for control over the order of execution, minimizing latency in data flow between subclasses.

See section “Overview” in *LibSched Programmer's Reference Manual.*

LibTime provides the *real-time millisecond clock* (set to zero at program start-up) which is the most accurate clock available, if needed for necessary system calls and error correction. A computationally less expensive function is also available if a few milliseconds of inaccuracy are acceptable.

The *internal simulation millisecond clock*, also provided by libTime, runs in real time by default. However, this clock can be used to achieve faster- or slower-than-real-time simulation. The simulation clock is advanced prior to every function invoked by the scheduler and remains constant throughout the resulting function thread.

In order to represent time in the shared PO database, the *shared simulation millisecond clock* (set to zero when the PO database is cleared or "New Scenario" is selected on the GUI) provides a common basis of measurement. LibPO maintains this basis through the heartbeat packets sent by all PO participants. In this way, time is synchronized for all ModSAF applications on different hardware platforms using the same PO database ID. The primary use of this time value is for scheduling of the HHour class and for freezing/resuming the simulation.

See section “Simulator Present PDU” in *LibPO Programmer's Reference Manual.*
See section “po_time” in *LibPO Programmer's Reference Manual.*

The internal simulation clock can be linked to the shared simulation clock to achieve a *distributed* faster-, slower-, or at-real-time speed simulation. (Note: Non-SAF systems, such as manned simulators, do not generally have non-real-time modes, so the use of time altered simulation is somewhat limited.)

Finally, the *wall clock* is used only on the GUI. It is accessed using Unix built-in functions (e.g. `time()`, `gmtime()`, and `localtime()`).

3 User Interface Architecture

The ModSAF Graphical User Interface (GUI) expedites the introduction of new functionality into the ModSAF system. In addition, it allows the operator to create or delete vehicles (and other entities), task them with behaviors, modify their reactions, and observe the outcomes. In order to provide the most consistent and easy-to-use interface, the ModSAF GUI meets the following specifications:

- The interface is modal. It must be in only one mode (e.g selection mode, creation mode). The activation of a new mode deactivates the previous one, limiting what the user can do at any point in time. This limitation helps to reduce complexity.
- The interface is tiled. It does not rely on pop-up windows for most interactions (pop-up windows make it difficult for the user to focus on the map display). Additionally, each of the different areas of the screen (e.g. editor window, tactical map) can be resized to use more or less space.
- The interface is extensible. It is easy to add a new software module's functionality to the user interface.
- The interface presents information in a consistent manner. User-input takes a limited number of forms.
- The interface complies with DoD standards and style guidelines.
- The interface is data-driven. It allows modification and specialization without recompiling.
- The user interface is reusable. It enables development of projects that need similar functionality.
- The interface is interoperable with a normal interface environment. It should act like any other window on a virtual desktop.
- The interface is portable. It should be compatible with windowing systems on most workstation platforms (X and Motif are the only options).
- The interface uses system resources efficiently. It is possible to run the user interface and simulation processes on a single computer.
- The interface is constructed automatically from the software definitions of available missions.
- The interface provides a structure that explains unit behavior to the user.
- The interface complies with the software architecture design methodology laid out earlier.

3.1 GUI Architecture Services

The ModSAF GUI is constructed using various service models. The use of service models ensures that consistency is maintained since all interactions with the user must pass through the same architectural support layers. Most importantly, it ensures that new modules can be added to the user interface without requiring changes to existing modules.

- LibSAFGUI is the lowest-level service library that provides the layout and utilities for control of the user interface tiles (menu bar, tactical map, mode buttons, editors, help lines, and message logs).

  See section "Overview" in *LibSAFGUI Programmer's Reference Manual.*
- LibPrivilege allows the password protection to be applied to different parts of the system.

  See section "Overview" in *LibPrivilege Programmer's Reference Manual.*
- LibTactMap is a tactical map service that manages the display of terrain features (i.e. hypsometric tinting, water, roads, trees, buildings, pipelines), graphical control measures, intervisibility lines/areas, and simulated entities and organizations.

  See section "Overview" in *LibTactMap Programmer's Reference Manual.*
- LibPVD provides tools to control the Plan View Display. It also displays events on the network, such as the appearance of vehicles and weapons fire.

  See section "Overview" in *LibPVD Programmer's Reference Manual.*
- LibBGRDB, the Battlefield Graphics Database library, is used by libTactMap to draw vehicles as either pictures or as military icons.

  See section "Overview" in *LibBGRDB Programmer's Reference Manual.*
- LibSensitive manages mouse-sensitive objects in an X windows environment. A call to libSensitive identifies which window in the user interface is sensitive, and which callback routines to invoke when various X events are detected in that window.

  See section "Overview" in *LibSensitive Programmer's Reference Manual.*
- LibEditor gets the input from the user, builds the graphical interfaces from data files, and translates input data into internal structures.

  See section "Overview" in *LibEditor Programmer's Reference Manual.*
- LibTaskEdit manages task and task frame editing.

  See section "Overview" in *LibTaskEdit Programmer's Reference Manual.*
- LibParmEdit manages the editing of model parameters at run-time.

  See section "Overview" in *LibParmEdit Programmer's Reference Manual.*

3.2 Automated GUI

The ModSAF graphical user interface architecture provides the following capabilities:

- Editing common graphical parameters (lines, areas, points, and text).
- Displaying unit organization and hierarchy.
- Setting up and assigning task frames and missions.

However, the bulk of the interface is devoted to allowing the user to specify task and model parameters at run-time. This interface is generated *automatically* from data files that describe each task and model. There is a one-to-one correspondence between parameters in a data file and the *editables* (a set of X widgets that correspond to one field on the editor).

The data files describe the data representation or memory layout of the parameters, the meaning of the various fields (e.g. speed, angle), and the default values used for each task or model.

This approach saves time and effort since no user interface programming is necessary when defining new tasks. Yet, if a customized interface was desired, one could be programmed.

4 Simulation Architecture

The ModSAF simulation architecture comprises the parts of the system that perform simulation of physical processes, and the first level of control over those processes. In order to maximize the number of entities that can be simulated at one time, only those features that are externally observable or significant to other simulation exercise participants are simulated.

One impact of this approach is to blur the distinction between physical subsystems and the humans who control them. For example, the aircraft dynamics simulation is guided not by flap and rudder settings, but by a desired heading. The transfer of abstract commands into physical outputs is handled within the same module as the rest of the dynamics and kinematics simulation. This allows the closed-loop effects to be observed, without the expense of simulating the internal processes.

The goals of the simulation architecture are:

- Add new simulation modules with a minimum of effort.
- Allow the simulation modules to be easily configured in arbitrary combinations within each simulated entity.
- Separate behavioral tasks from the physical systems they control and the utility models they simulate to maximize the reuse of software and minimize the expense of adding new modules.
- Provide environmental models to simulate aspects of the environment that are perceptible and tactically significant.
- Provide a generic interface for the simulation of the various physical components of each entity.
- Make the most efficient use of computational resources to maximize the number of vehicles simulated.
- Provide support for automated navigation at the architectural level to allow task developers to focus on higher level issues.
- Provide dead reckoning capability by approximating vehicle movement to reduce DIS network traffic.
- Support models of varying fidelity and allow selection of fidelity at run-time.

The architecture meets these goals by augmenting the ModSAF software design methodology with a set of conventions and tools as described in the following sections.

4.1 Physical Models

Since the definition of *physical* in ModSAF often includes the lowest level of human interaction, specifications of physical models must be explicit. In ModSAF, the following are considered. *physical* processes:

Hull Simulation

A hull simulation uses a desired heading and velocity as input; then generates the location and velocity of the entity as its output. Hulls include ground and air vehicles, Dismounted Infantry (DI), and missiles. A single entity can only have one instance of a hull model.

See section "Overview" in *LibHulls Programmer's Reference Manual.*

Turret Simulation

A turret simulation uses desired orientation as input; then simulates the rotation of the turret. Currently there is one turret model that can support up to four instantiations per vehicle.

See section "Overview" in *LibTurrets Programmer's Reference Manual.*

Sensor Simulation

Sensor simulations filter the ground truth information available in the DIS database abstraction and present a clouded view of the world for processing by tasks. The input is one focus of attention. The simulation of human ability to interpret the information is included as well. Examples of sensors include visual, radar, and thermal.

See section "Overview" in *LibSensors Programmer's Reference Manual.*

Weapons Simulation

Procedures such as loading, aiming, and firing weapons are simulated within the weapons simulation process. The time used by operators to perform actions such as loading or acquiring a target is included in the weapons simulation, as are models of human ability to use those weapons effectively. Weapons models include ballistic guns and missile launchers. Multiple instantiations per vehicles are supported.

See section "Overview" in *LibGuns Programmer's Reference Manual.*

4.2 Utility Models

The following are examples of low-level utility models. There is only one instance of this model per entity, and the events are network-driven.

Detonation

Detonations are detected due to the proximity with other entities. In other words, detonation occurs if two vehicles pass each other within a given distance (such as a missle being fired over a tank). On the other hand, if these vehicles pass closely by one another outside of a given distance, then it is declared a near-miss.

See section "Overview" in *LibDetonation Programmer's Reference Manual.*

Fire Control System

The fire control system provides an abstraction around a collection of guns for ModSAF vehicles. The interface allows weapon launcher commands to be specified as fired munitions.

See section "Overview" in *LibFCS Programmer's Reference Manual.*

Damage Simulation

The effects of weapons fire, wear and tear, and misuse are considered damage models. These models are influenced by aspects of the simulated environment, rather than behavior simulations. The damage models are based on standard probability of kill tables residing in reader files.

See section "Overview" in *LibDFDam Programmer's Reference Manual.*
See section "Overview" in *LibIFDam Programmer's Reference Manual.*

Collision The collision utility model detects collisions with other network entities. Collision is determined for both nearby intersections, as well as for fast-moving vehicles that travel long distances in every tick. In the latter case, a ray from the vehicles's old position to the new position would check for intersections.

See section "Overview" in *LibCollision Programmer's Reference Manual.*

Communications Simulation

The transmission and receipt of radio or other communications is simulated within ModSAF. Formatted text radio messages are used to coordinate actions between vehicles as well as to inform supervisory control processes (including the user interface) of events.

See section "Overview" in *LibArtyRadio Programmer's Reference Manual.*
See section "Overview" in *LibGenRadio Programmer's Reference Manual.*

Supplies This facility manages the bookkeeping for supplies. It maintains a list of munitions, knows the amounts of these munitions, decrements its store appropriately, and can explicitly have its quantity of supply levels set.

See section "Overview" in *LibSupplies Programmer's Reference Manual.*

4.3 Behavioral Models

Behavioral models or *tasks* are divided into two categories:

1. *Vehicle* tasks are assigned to *individual* vehicles. Common examples include move, collide, terrain, search, spotter, enemy, assess, targeter, stop, orbit, etc.
2. *Unit* tasks are assigned to *collective* units. A unit is either a platoon, a mixed platoon, a company, a battalion, or a flight. Some examples include assault, actcontact (actions on contact), enemy, poccpos (preparatory occupy position), halt, targeting, march, attack, traveling, hover, flyrte (fly route), etc.

For more information on any of the behaviors listed above, see their corresponding Programmer's Reference Manual. Note: the task's prefix indicates which kind of unit it belongs to. For instance, information on vehicle move is located in libvmove. Likewise, information on unit assault is located in libuassault, mixed halt in libumxhalt, company march in libucmarch, RWA and FWA tasks such as fly route and hover in liburwahover and libufwaflyrte, etc.

4.4 Environmental Models

To simulate environmental effects on the battlefield, environmental models, such as tactical smoke, can be added to ModSAF. (This is known as the the Dynamic Virtual World (DVW) Project or Phenomenology Project, sponsored by the Advanced Research Projects Agency (ARPA) and the U.S. Army Topographics Engineering Center (TEC).)

The environmental architecture is model independent, open-ended, and easily reconfigurable. Key components that help to implement a model include:

Query Interface

Hides the internal details of the models from the application (in this case, ModSAF) and combines the outputs from multiple models into one single output.

Effect Interface

Allows the application to create an effect without knowing which model is producing it.

See section "Overview" in *LibEnvironment Programmer's Reference Manual.*

4.5 Generic Model Interfaces

New weapons systems, sensors, etc. are constantly being introduced into the ModSAF system. If general mechanisms can be defined to interact with physical systems, then the incremental cost of adding new systems can be minimized. Without pre-defined interfaces, every new physical simulation module would require re-engineering of the higher layer software modules that access it.

To minimize the impact on higher layer modules when a new lower layer module is introduced, a set of *Generic Model Interfaces* (GMI) has been defined. These interfaces define all the necessary input and output routines that tasks need to access. The use of GMI make it possible for a single task to control a wide variety of similar physical systems. For example, a single task can control the movement of all ground vehicles, infantry, and many types of air vehicles through the use of LibHulls.

See section "Overview" in *LibComponents Programmer's Reference Manual.*
See section "Overview" in *LibHulls Programmer's Reference Manual.*

4.6 Near-Term Navigation

ModSAF contains an architectural service that provides solutions to navigating a vehicle in the near-term, including obstacle avoidance, road following, and moving in formation. The service is libMoveMap, and it is based on the principle of efficiently combining near-term goals and constraints into a single map.

Higher software layers provide the near-term goals and various movement constraints as input; libMoveMap generates near-term plans to achieve those goals. A variety of movement goals, such as cross country route following, road following, cross country station keeping within a unit, and traveling with even spacing along a road can be achieved.

LibMoveMap can plan a course through space and time for the vehicle to accomplish its near-term goal, avoiding obstacles while obeying physical constraints. When unable to achieve the goal outlined by the higher layer software, libMoveMap indicates it is "stuck". It is the responsibility of the higher-level routines to re-define the goals. Finally, output (in the form of desired velocity vectors) is passed on to hull simulation.

For a detailed description of libMoveMap and the algorithms it uses,

See section "Overview" in *LibMoveMap Programmer's Reference Manual.*

For more information on collision detection, route manipulation, and other vehicle movement libraries:

See section "Overview" in *LibCollision Programmer's Reference Manual.*
See section "Overview" in *LibRoute Programmer's Reference Manual.*
See section "Overview" in *LibRouteMap Programmer's Reference Manual.*
See section "Overview" in *LibLocalMap Programmer's Reference Manual.*
See section "Overview" in *LibVMove Programmer's Reference Manual.*
See section "Overview" in *LibVTerrain Programmer's Reference Manual.*
See section "Overview" in *LibUTraveling Programmer's Reference Manual.*
See section "Overview" in *LibVFlwRte Programmer's Reference Manual.*

4.7 Terrain

ModSAF provides realistic terrain reasoning capabilites for military mission execution. Given user-specified criteria, terrain features, and other constraining parameters, vehicles plan their tasks and react appropriately, while avoiding static and dynamic obstacles. These terrain reasoning capabilities include crossing rivers, finding cover and concealment, determining safe overwatch hop points, and performing contour flight.

See section "Overview" in *LibTeReason Programmer's Reference Manual.*

Terrain features are represented by Compact Terrain Database Bases (CTDB). The CTDB representation allows storage of different classes of terrain objects, uses various compression techniques to reduce memory requirements, and is modeled using a regular grid of elevations, as well as areas of more precise microterrains. In ModSAF, libCTDB provides the facility to

1. Read the database
2. Maintain information such as size and position of the database
3. Access elevation and soil type
4. Place and orient vehicles
5. Calculate intervisibility
6. Calculate radar clutter
7. Display terrain graphics such as contours and hypsometric maps

For a detailed description of libCTDB and the algorithms it uses, See section "Overview" in *LibCTDB Programmer's Reference Manual.*

4.8 Dead Reckoning

Dead Reckoning (DR) is a vehicle movement method that reduces DIS packet traffic, thus increasing the number of DIS network entities that can be simulated in a single exercise. In a simulation where moving vehicles interact, up-to-date information is needed regarding the location of the vehicle. For instance, manned simulators need current information at the visual system update rate so that the graphic images move smoothly. This update rate ranges from 7 to 50 times per second. For this information to be available, the other simulators on the network have to retransmit their vehicles' position, orientation, etc., at the same frequency.

Dead Reckoning factors in that entities in the real world do not move from moment-to-moment in a totally random fashion. If a vehicle is moving in one direction at a particular speed, a few milliseconds later it is still moving in approximately the same direction and speed. Any abrupt alteration of this movement (such as when a tank turns at a sharp curve in the road) takes time to occur.

Dead Reckoning assumes that if a vehicle notifies the other DIS simulators its location and destination, they can then predict where the vehicle will be in the short term without further notification. If the vehicle starts to deviate substantially from where the other DIS simulators expect it to be, it will notify them of its true location and destination. This is Remote Vehicle Approximation (RVA). Under the SIMNET protocol, there is one simple RVA algorithm in which the current vehicle position would be extended linearly on its last direction of travel at a constant speed. If the simulator that is responsible for creating the entity finds that the orientation or position of the vehicle is off by more than a predetermined threshold, it sends another update. The DIS protocol can utilize more complex algorithms that consider acceleration and other factors to even further reduce network traffic, while still using the concept of thresholding and retransmission.

See section "Overview" in *LibDR Programmer's Reference Manual.*

4.9 Model Fidelity

There is a constant trade-off between model fidelity and computational cost. Depending on the resources available and how critical the accuracy of the simulation is, model fidelity can be altered through the data files to achieve optimal simulation results. Model fidelity is varied by changing various parameters, such as the vehicle tick rate or movement planning time. A classic example of the application of varying model fidelity is collision detection. For example, ModSAF has two models of collision detection, "high," and "low." Low fidelity treats objects like spheres for detection of collisions and like cubes for computation of the collision point. High fidelity treats objects like rectangular solids (parallelepipeds) for both the detection of a collision and the computation of the collision point.

See section “overview” in *LibCollision Programmer's Reference Manual.* The ModSAF architecture also supports dynamically controlled model fidelity; the ability to change models, or parameters of models at run-time during execution. An example of dynamically controlled model fidelity is changing the parameter from "high" to "low" during execution, if there is an increase in system load.

5 Command and Control Architecture

The ModSAF system has an architecture for representing command and control (C2) information. It effectively simulates human behavior and military command and control. It consists of a set of Persistent Object (PO) class definitions and algorithms that dictate how objects of those classes are treated by the user interface and simulation. The parts of the existing ModSAF command and control system belong to one of two categories:

1. The architectural framework used for command and control.
2. Implementation of a specific approach within this framework.

The C2 architecture meets these specific goals:

- Ensures that the user can construct arbitrarily complex missions that may include pre-planned contingencies and task reorganization (adjusting the command hierarchy as the mission proceeds).
- Ensures that the user can subsequently change the initial mission definition after the assignment (*frago* capability).
- Ensures that the user can override choices made by the simulated units at run-time.
- Provides a way to express the type of mission that the units are executing (formerly called a *combat instruction set*).
- Provides a general representation for unit and individual behavior without mandating a specific approach to behavioral representation.
- Provides the option of representing battlefield uncertainty (radio transmission models, insubordination, fog of war, etc.).
- Provides a structure that allows the user interface to describe unit behavior to the user.
- Provides a direct mapping between the architectural structure and the user interface.

The following sections detail the architectural framework, and explain the initial ModSAF implementation of a command and control system.

5.1 Framework

The concept of a *task* is the foundation of the ModSAF command and control framework. A task is a behavior performed by an individual on the battlefield. Tasks can:

- Be done as a unit (company road march)
- Directly control a physical system (drive toward a waypoint)
- Achieve a mission objective (attack an objective)
- Be done continuously, independent of the mission (scan for enemy)
- Be representations of actual battlefield behavior (run for cover)
- Be implementation details of the simulation (arbitrate between several possible alternative actions)

A task is represented in the Persistent Object database in three parts:

1. The task *model* number is a unique number that tells the simulation software the body of code to use to execute the task. The specificity of each model is an implementation decision.
2. The task *parameters* express different options available in executing the task. For example, the route to follow, or the objective to attack are task parameters. The degree to which a task is parametrically defined is another implementation decision. The format of the parameters is defined by the task implementor.
3. The task *state* uses the task model during execution. This public version of the task state holds information that is shared for use by the user interface, for data analysis, or for fault tolerance. The format of the state is also defined by the task implementor.

See section "Task Class" in *LibPO Programmer's Reference Manual.*

While the architectural framework defines a storage place for these items, it makes few requirements about the content or format used to represent this information.

The remainder of the architectural framework expands upon the idea of a task. There is the *unit hierarchy* that represents the different roles played by individual vehicles; the *task frame* that allows related tasks to be grouped; *task frame stacks* that allow task frames to be suspended, resumed, or overridden; *enabling tasks* that are used in the construction of complex missions; and the *task manager* that comprises the algorithms used for execution of tasks.

See section "Overview" in *LibTask Programmer's Reference Manual.*

The following sections describe the remaining tasks.

5.1.1 Unit Hierarchy

The *unit hierarchy* is represented in the Persistent Object database via Unit Class objects. Each SAF vehicle in the simulated environment has a corresponding unit in the PO database. The organizational layers above these individuals (platoons, companies, flights, etc.) are represented by units as well.

See section "Unit Class" in *LibPO Programmer's Reference Manual.*

The units in the database are organized as follows:

Organic Organization
: This is the initial organization. It is useful to the user as commander since it gives each vehicle a recognizable handle. The organic organization does not change often; perhaps only after major battles.

Task Organization
: This is per mission unit organization. The hierarchy of command can be set up before each mission and can be programmed to change as the mission proceeds.

Functional Organization
: This is the frequently changing organization that results from executing different tasks. For example, during bounding movement, a small unit may break into different functional subunits. Each of these subunits is represented, and can execute tasks. The user can see this functional organization, so that downward control remains available.

See section "Overview" in *LibUnitOrg Programmer's Reference Manual.* All tasks are executed by individuals on the battlefield; each unit has a vehicle that is responsible for executing its tasks. The set of units (for which an individual is responsible) is referred to as *roles.*

5.1.2 Task Frame

Task Frames (similar to Combat Instruction Sets) group a collection of related tasks that run simultaneously. A task frame that represents a phase of a mission (road march or attack) is defined in data files by a set of task names, and the parameters used by those tasks. These task parameters are defaults that a developer can edit to customize the operation of the tasks.

See section "Task Frame Class" in *LibPO Programmer's Reference Manual.*
See section "Overview" in *LibTaskFrame Programmer's Reference Manual.*

In addition to allowing the user to set up task frames to define a mission, the simulation software can assisgn its own tasks and task frames at runtime. For example,

- High-level unit tasks, such as those executed by a battalion, can build task frames for subordinates to execute. In this way, the downward flow and refinement of commands are simulated. These frames can be examined at run-time or during data analysis after the exercise. It is also possible to modify the task parameters in these frames at runtime.
- Reactive tasks can construct and execute task frames to deal with changes in the battlefield environment. For example, a company of tanks on a road march can push a new task frame in response to driving into a minefield. The new frame can give a different set of tasks to execute, and the parameters of these tasks can be customized to the specific environment (choose a safe path out, etc.) at runtime by the reactive task.
- Unit tasks can create tasks for assignment to individuals. The unit tasks then monitor the progress of these created tasks, and adjust to their parameters as needed. The ability to define behavior hierarchically simplifies task programming.
- Each unit has a background frame in which it can place tasks that continue to run after the current tasks are completed. For example, if a battalion task sends a scout platoon ahead, it can then create a task in the background frame to monitor the feedback of that scout platoon.
- Each individual has a single frame for background tasks that run continuously, regardless of the mission, and as a place for units to place new tasks.

Assigning a task frame to a unit is a two-step process.

1. The task frame is marked with the unit ID, indicating who is to execute the frame.
2. The task frame is pushed onto the unit's task frame stack when the machine simulating the vehicle detects a change in the task frames.

Although the ModSAF software currently pushes every received task frame, the decision to push the frame can be contingent upon battlefield uncertainties (did the message get through, does the commander of the unit choose to obey the order, etc.). This describes the method of *layering uncertainty on top of ground truth*. During the "1989 SAF Proof of Principle" exercise, it was shown that the ability to eliminate the uncertainty of battle is mandatory in the initial development of behavioral algorithms. Once the fundamental behaviors are in place, the uncertainty of battle can then be added, and the behaviors can be enhanced to deal with this uncertainty.

5.1.3 Task Frame Stacks

As a mission proceeds, a unit sometimes needs to switch to a different task frame for a while, then resume the original task. Similarly, the user may want to override a pre-planned behavior temporarily. Both needs are addressed by *task frame stacks*. Rather than a unit executing a single frame, it executes a stack of frames. The topmost frames are currently active, and the rest are suspended. New task frames can be pushed on the stack as the result of a reactive task (such as reacting to a minefield), or at the user's request (in response to an unexpected event). As an optimization, each task frame in the stack is marked as either *opaque* or *transparent*. The tasks in each transparent frame at the top of the stack are merged with the tasks in the topmost opaque frame. This allows tasks to be layered over the original mission framework without requiring that all unaffected tasks be duplicated into the topmost frame. The use of transparent task frames simplifies the following user interface needs:

Preprogrammed Instructions

Ground missions can generally be defined via a series of isolated task frames that are executed in sequence (road march, prebattle march, attack). Air missions, however, are more easily defined via a single task frame, with occasional modifications. This latter method is achieved by attaching *instructions* to geographic or temporal features. Each instruction changes the parameters of some subset of mission tasks. This method of mission construction is useful in the ground domain as well ("change formation when the vehicle crosses this line"). (Note: Although this is supported by the architecture, there has not yet been a requirement to put this control in the user interface.)

These instructions are implemented by placing the modified task into a transparent frame which is placed at the top of the stack when the conditions for the instruction have been met.

Temporary Runtime Modifications

As a mission proceeds, the user may want to temporarily change the mission definition. When this override is a change to a parameter of a currently executing task, a transparent frame can be placed at the top of the stack, containing only the impacted task. All other tasks continue to operate normally. When the user want to return to the original behavior, the transparent frame is removed.

5.1.4 Enabling Tasks

Often, certain contingencies can be predicted. To specify alternate missions for those cases, *enabling tasks* (defined the same way as other tasks, with a model number, parameters, and state) are used to express the most complicated contingencies.

A *mission* is a set of task frames that are linked to form a sequence. In a simple mission, each task frame specifies one task in the previous task frame that must complete before the subsequent frame starts. For example:

```
---------------------------------------------------------------

 Road March Task Frame             Attack Objective Task Frame

[ follow route task       ]<--.  [ attack task                ]
[ targeting task          ]   |  [ targeting task             ]
[ minefield reaction task ]   |  [ hasty retreat reaction task ]
[ ...                     ]   |  [ ...                        ]
                              |
< Enabling Task: none     >   '- < Enabling Task: follow route >

---------------------------------------------------------------
```

In this example, the Attack task frame (which comes after the Road March task frame) refers back to the follow route in the road march as its enabling task. When the follow route is finished, the Attack task frame is enabled (pushed onto the stack, in place of the road march).

A more complicated example is the execution of a subsequent task frame that is not contingent on completing an executing task. It can be triggered by seeing an enemy unit, passing a point, time elapsing, etc. These are *predicate* functions. They are defined as enabling tasks. These tasks are unusual in that they are only executed when the task frame that depends on them is *not* running.

For added flexibility, the architecture combines the results of enabling tasks in arbitrarily complex logical expressions. For example,

```
"attack if you spot the enemy AND you cross phase line beta."
```

5.1.5 Task Manager

The *task manager* is the software (spread over a few libraries) that implements the architectural framework algorithms. Each task model, at application startup, registers itself with the task manager and specifies the following components:

Model Number

The task manager uses the model number (protocol constant) to make a mapping between a task in the Persistent Object database, and the software which implements that task.

Entry Points

Each task provides the task manager with the entry points (function pointers) for task functions that the task manager needs to invoke:

`Params`	The task manager calls this function when the task parameters change. This frees the task implementor from monitoring the PO database.
`Predicate`	This function is called when the task is an enabling task. It returns a `true` or `false` value according to whether its conditions have been satisfied.
`Start`	This function is invoked when a task is started, or restarted after having been suspended (see `Suspend` below).
`Tick`	This function is invoked once each vehicle tick while the task is running. It lets the task update its internal state over time. Purely event-driven simulations may not use this entry point.
`End`	This function is invoked when the task is forcibly ended, such as when the task frame in which it resides is popped off the task frame stack.
`Suspend`	This function is invoked when a task that has not ended stops temporarily. This generally happens when an opaque task frame is pushed onto a task frame stack.

Shared Data Size

Although the task manager does not need to know the specifics of each task's implementation, it does need to know the size of the parameters and state used by the task for network operations. The task manager simplifies task writing by automatically updating the PO database when tasks change their states.

Before and After Lists

The tasks executed by an individual can be interdependent and rely on input from other tasks (which make up its *before* list). Tasks can also provide output to other tasks (which make up its *after* list). The task manager ensures that the tasks are ticked in an order that minimizes latency in the information flow.

The task manager, after receiving this information from each task model, decides which tasks to run. For more information on the algorithm that the task manager uses,

See section "Overview" in *LibTaskMgr Programmer's Reference Manual.*

5.2 Implementation

The following describes implementation decisions made during the development of the ModSAF command and control system. The system utilizes a common software method for encoding task models through the use of finite state machines and arbitration methods.

5.2.1 Task Data Structure

The software representation of a task is a Finite State Machine (FSM) that has several task specific states, and always the **ended** and **suspended** states. The state machine implementation and the data structures it operates on are referred to as a task model.

A task model interacts with data that can be

- *Shared* – in the Persistent Object database, where any application program on the network can examine the data
- *Local* – only available in the simulation process that is simulating the vehicle that is executing the task
- *Public* – defined in public headers so that other software libraries can use the data
- *Private* – defined in private headers so that the data is only available to the model

The four abstract data types are as follows:

System Parameters (Shared, Public)

System parameters (`<task>_PARAMETRIC_DATA`) control how a task performs. These parameters are set for each kind of vehicle (F14 vs. M1, etc.) to specifically guide execution of the general task in appropriate for that entity. These parameters are also used to tune execution, or to allow system-level field modification without the need for programming or compilation. System parameters originate within data files, but can be modified *re-configured* at runtime via the Persistent Object database.

Task Parameters (Shared, Public)

Task parameters (`<task>_PARAMETERS`) control the execution of the task in the context of a single mission. These include mission parameters such as what route to follow, or the Rules of Engagement.

Shared Task State (Shared, Private)

Shared task state (`<task>_STATE`) is maintained by an executing task. It includes the primary state machine state variable as well as any specific state information (current route point or a list of detected targets) that needs to be shared by the user interface, for data analysis, or for fault tolerance needs. This state must not change frequently since every change leads to additional network traffic.

Local Task State (Local, Private)

Local task state (<task>_VARS) is also maintained by an executing task. It will generally contain either more specific versions of the data in the shared task state or more efficient internal representations of the same information. Local task state may change frequently without any performance detriment; however, this information is not available for monitoring, analysis, or fault tolerance needs.

Enough state must be shared to enable the unit leader (who assigned the task) to monitor the task progress. Also, enough state must be public to enable other individuals or unit leaders to take over execution of the task with no observable difference. This takeover can be required when a machine fails or when the unit level task for the company commander is destroyed, and another vehicle in the unit must take over the task. The programmer needs to encode the task ensuring that the public state does not change frequently because of the bandwidth limits in the communications media. A task is described in the Persistent Object protocol by an object of the class Task, and another object of the class TaskState (shown below with padding omitted):

```
type TaskClass sequence {
    model        UnsignedInteger (32),
    frame        ObjectID,
    state        ObjectID,
    suspended    Boolean,
    stateReuse   Boolean,
    primary      Boolean,
    refcount     UnsignedInteger (8),
    size         UnsignedInteger (16),
    data         array (size) of UnsignedInteger (8)
}

type TaskStateClass sequence {
    refcount     UnsignedInteger (8),
    taskSequence UnsignedInteger (8),
    size         UnsignedInteger (16),
    model        UnsignedInteger (32),
    data         array (size) of UnsignedInteger (8)
}
```

The task object holds the parameters of the task while the state object holds the runtime state.

When a task object is created, it contains a task model (a reference to the body of code and data that execute the task in the context of the simulation), the frame in which the task resides, the state object that it uses, the number of referenced parameter objects (such as the control measures that could influence the execution of this task), and task data. Any attribute of a task can be changed after initial creation (although the size is generally a constant). The *task* data holds the parameters for the task while the *task state* data holds the state of the task. These are interpreted by the task state machine that implements the task. Initialization of the parameters and state of a task is the responsibility of the unit leader who assigns the task (since the format of the state is private, the body of software that initializes state resides within the task library, and is invoked by the unit leader). An executing task updates the shared state whenever appropriate. As an example, the shared representation of a tank's state, while it is running the spotter task that scans for objects, includes a list of detected objects and their presumed alignments. This state is updated periodically, at a rate specified in the system parameters of the spotter model. Also included in the system parameters for the tank's spotter task is a list of sensors that needs to be read and manipulated. If other tasks within the tank need a list of the tank's spotted vehicles, those tasks can call the LibSpotter function, `spotter_get_spotted`, which will return the local representation as a list. Other applications, such as the user interface, can probe the shared representation of the tank's spotter task by passing the entire task to the LibSpotter function, `spotter_get_spotted_from_state`.

5.2.2 AAFSM Format

Behavioral tasks are implemented using Asynchronous Augmented Finite State Machines (AAFSM). They are *Asynchronous* because they generate outputs in response to events in the simulated environment. They are *Augmented* state machines because they influence and use many variables other than their state variables. Rather than implementing the state machines as switch-case constructs, the state machines are defined as modified state transition tables. A preprocessor utility, '`fsm2ch`', reads the AAFSM format and translates it to C code that can then be compiled. In addition, the preprocessor can generate transition diagrams as fig source that can be reviewed using '`xfig`' and printed using '`fig2dev`'. The '`fsm2ch`' utility also has limited code-generation facilities, to simplify writing tasks that control other tasks (unit behaviors), and tasks that generate task frames (reactions).

The following is an example of a simple task that monitors fuel usage, and reacts to low fuel levels by pushing a "Return to Base" task frame:

```
static int32 veh_bingofuel_reached();

'
ubfuel UBINGOFUEL_PARAMETERS UBINGOFUEL_STATE UBINGOFUEL_VARS

SUBTASK: Unit_Return_To_Base rtb_task SM_URTB URTB_PARAMETERS
 URTB_MAX_TASK_REFERENCES urtb_init_task_state
{
    subtask->base = parameters->base;
    subtask->speed = parameters->speed;
    subtask->altitude = parameters->altitude;
}
END_SUBTASK

SUBFRAME: Return_To_Base rtb_taskframe TIIPopOpaque opaque destroy
INCLUDE: Unit_Return_To_Base
END_SUBFRAME
‘
                tick()
‘
                        params()
‘
START
‘
                private->last_fuel_reading = 0.0;
                switch (state->state)
                {
                  case evaluating:
                        break;
                  case ended:
                  default:
                        ^evaluating;
                }
‘
evaluating
‘               tick
                if (veh_bingofuel_reached(vehicle_id, private,
                                          parameters, state, task_entry))
                {
                    SPAWN SELF Return_To_Base
                }
‘                       params
                        private->last_fuel_reading = 0.0;
‘
END
‘

‘
```

After specifying data structures that will be used by the task, the FSM defines the frame that it may push, and the tasks that go into that frame (in this case, there is only one task). Next, the events that this task supports (in this case, only the standard tick, and params events) are declared. The FSM specifies the conditions when the task is started (or resumed, or migrated from another simulator). And finally, the FSM defines the behavior that is called for by each event, in each state (this example task has only one state).

For details about the syntax and structure of FSM files, See section "AAFSM Code Generator" in *LibTask Programmer's Reference Manual*. The C source code generated from '`.fsm`' files is compiled and linked with supporting software to make a task model library. This model is linked into the application program to become a DIS DB object subclass. Tasks that need to be driven by events other than built-in functions `tick` and `params` define the public interfaces to these functions in the task library public header file, and put calls to these functions elsewhere in the simulation software.

5.2.3 Arbitration

Arbitration is balancing competing objectives with limited resources. Each task within a vehicle makes its recommendations for movement, gunnery, etc., and ultimately a decision must be made. This situation is approached in three ways (all of which can be supported by the ModSAF task management software):

Context-Free Arbitration

The needs of the various tasks are expressed in a uniform way, and various priorities and weighting schemes are applied in a transfer function to yield one output to each actuator. This approach has the advantage that the arbitrator does not need to change when tasks are added; however, all the weights and priorities generally do need to be modified for each added task. As the number of tasks grows large, finding appropriate values can become difficult.

Context-Rich Arbitration

Each task determines its preference and expresses it in a way that provides all the context used to make that decision. The arbitrator then merges these inputs, using whatever context is necessary, and makes an informed decision regarding how to proceed. This approach requires that the arbitrator be modified for each added task; however, the decision making rules are simpler to define than abstract weights so new input can be incorporated with less work when the number of tasks is large.

Mutual Arbitration

Each task is aware of the requirements of the other tasks and the system is driven by consensus. Such systems become complex when the number of tasks is large since each task may need to be changed when new tasks are added. Also, as with context-free arbitration, isolating and eliminating undesired behavior can be difficult.

A task arbitrator must decide how to take the recommendations from a group of tasks and produce a single output that is sent to a physical subsystem (sometimes called an "actuator") or another task. The goal of the architecture is to be flexible to support different conflict resolution schemes. This enables experimentation with different schemes and allows the implementation of the developer's own schemes within the ModSAF system. The choice of how to combine competing recommendations from different tasks is not of central concern to the discussion that follows; rather, the focus is on the description of the flexible architecture that surrounds the conflict resolution mechanism.

Previous SAF systems used mutual arbitration, and some have experimented with context-free arbitration. For ModSAF, context-rich arbitration was initially chosen because it was able to build a sophisticated and robust SAF quickly. However, the data flow issues involved in having a few arbitrators trying to understand the full-context desires of many tasks make this approach somewhat impractical. For example, the full contextual environment of collision avoidance includes every obstacle in the local configuration space. Understanding this context would require the arbitrator to incorporate all collision avoidance algorithms.

Currently, the software uses mutual arbitration, structured and supported by architectural enhancements to minimize the complexity problems. For the class of issues that need to be solved in a context-rich manner (such as ground vehicle movement), architectural support libraries (such as libMoveMap) are constructed to merge constraints from multiple sources, to achieve a goal specified by a single source. Such libraries provide context-rich arbitration of constraints but no arbitration of goals. Arbitration of goals is supported by libTaskPri (for Task Priority) that now executes as part of the task management algorithms.

See section "Overview" in *LibTaskPri Programmer's Reference Manual.*

When tasks are selected to run for a tick, but before their execution, certain tasks are selected to provide actuator output for that tick. The tasks specify at initialization their criteria for providing output so this is a mutual arbitration scheme. Tasks can have interdependencies (for example, "Task A can only provide output to Actuator 1 if Task B is not providing output to Actuator 2"), and the algorithms in libTaskPri choose a highest-priority combination of tasks that meet their individual output criteria.

The tasks provide output *directly* to the actuators when they are selected as the task to run. This has several advantages over the prior fully context-rich implementation:

- The algorithms for processing the contextual information that impacts decision making are distributed among the libraries that collect that information. This improves modularity and simplifies debugging.
- Feedback paths from the actuators are simple and direct. In contrast, other arbitration schemes provide no obvious method of feedback to reach the tasks that are controlling the actuators.
- Tasks are *aware* when they are in control. Without this kind of feedback, tasks cannot distin-

guish between failing because their output is inadequate versus failing because their output is being ignored. The two situations generally require different responses from the task (such as giving up versus waiting).

- Unnecessary computation is avoided since tasks which do not provide output know it a *priori*.
- Latency is reduced because the generation of actuator commands, the issuing of those commands, and the processing of feedback can occur within the same thread of execution.

However, there are some disadvantages:

- Complexity increases with number of tasks. Existing tasks may need to be changed when new tasks are added.
- It is sometimes difficult to isolate undesired behavior.

Although currently the software is arbitrating only for control of actuators, the architecture is designed to allow any critical resource to be defined as needing arbitration. This includes providing input to other tasks for multiple-level arbitration; arbitration between reactive tasks to determine which will be allowed to react when many want to; or even arbitration over critical computing resources within the context of a single entity.

As this support for mutual arbitration was added to the architecture, the underlying support for other arbitration schemes remained unchanged. During the initial implementation, the new scheme was implemented for arbitration over movement (the `hull` actuator) while the previous arbitration mechanism remained in place for weapon and sensor control.

6 Data Flow in ModSAF

The following feed-through view of the ModSAF software system describes the concepts discussed in previous chapters. It shows the flow of data through the primary components of the system – from user input down to the transmission of packets.

```
                               ---------------------
                              | Supervisory Control |
                               ---------------------
Create/Modify      ||                                     /\      visual
units,             ||       Graphical User Interface      ||      display
task frames        \/                                     ||
                               ---------------------
                              | Collective Behavior |
                               ---------------------
Create/Modify      ||                                     /\      events
tasks              ||            Tasks Interface          ||      and
                   \/                                     ||      polling
                               ---------------------
                              | Individual Behavior |
                               ---------------------
Initiate/Modify    ||                                     /\
internal           ||       Generic Model Interfaces      ||      polling
simulations        \/                                     ||
                               ---------------------
                              | Physical Subsystems |
                               ---------------------
Modify             ||                                     /\      events
outward state      ||          Service Interfaces         ||      and
                   \/                                     ||      polling
                               ---------------------
                              |  Support Services   |
                               ---------------------
```

Note: Throughout the following discussion, "you" is referred to as the user.

- Supervisory control is at the top. In ModSAF, this is primarily provided by the Graphical User Interface (GUI). You are presented with abstractions, such as creation and placement of objects on the battlefield, unit task organizations, and execution matrices. Thus, via the graphical interface, you can create or modify units and task frames.
- Units and task frames correspond to groups of entities and behaviors within the SAF simula-

tions. The tasks in the task frames, when executed together by the simulated entities, result in collective behaviors.

- The units (e.g. companies, platoons) examine the collective or group behaviors and assess the battlefield situation. Based on the information in a given scenario, the units create or modify individual tasks in the task frames via the task interface. The execution of these tasks by single SAF entities (the subordinates) results in individual behavior.
- These entities perform their individual behaviors by calling the Generic Model Interfaces (GMI). These calls, in turn, cause the physical subsystems to initiate or modify internal simulations of the entity behaviors as well as the physical effects of these behaviors.
- Using the interfaces provided by the service layers, the physical subsystems cause changes to the outward state of the vehicles being simulated and transmit this information over the DIS network.
- These changes are sensed by other physical subsystems when they poll the service interfaces or receive information about an event. This causes state changes in the internals of these subsystems.
- Individual behaviors query the generic model interfaces, and learn of the changes to the perceived state of the world. These tasks change their own state in response.
- Through polling or events, the collective behaviors observe the changes to the state of the tasks which represent the individual behaviors. They respond by changing their own state and possibly modifying the tasks they have created.
- The supervisory control observes these changes and reports them to you via visual status displays on the graphical user interface.

This feed-through view of the data flow outlines, *in general*, the sequence of actions and responses that are initiated during SAF simulation. However, this view may be misleading; information from lower-layer modules permeates *all* the higher-layer modules. In other words, higher-layer modules can utilize information not only from the layer directly beneath but from lower layers as well. For instance, the supervisory control process has access not only to the status of the collective behaviors, but also to the status of the individual behaviors and the state of the entire DIS database. Similarly, a vehicle executing a collective behavior can use information from its sensors (physical subsystems), in addition to the input it receives from tasks (individual behaviors).

To more clearly illustrate the inter-relationships of the software modules, the remainder of this chapter explores the data flow through several common operations associated with *creating a platoon*. Note: This example is *not* comprehensive – many actions performed by the libraries are not mentioned. You are referred to the specific library documentation for details of the exact operations of the system.

6.1 Creating a Platoon

The following is a general data and execution flow sketch of the activities performed by the ModSAF system when creating a platoon.

6.1.1 Selecting Create Mode

The following is a description of the events that occur when you click the mouse button on the interface, selection mode switches to creation mode, the creation editor is displayed, and control is transferred back to the scheduling service.

6.1.1.1 Mouse Click to Button Press

The SAFstation displays the graphical user interface. LibSAFGUI creates the layout of buttons, menu bars, editors, maps, etc. You are in supervisory control in the Select Mode (base or "arrow" mode). LibSelect manages the actions of the interface during the select mode. When you click the mouse on the create button (the blue tank icon button), a message is sent from the X window server process (a program that runs continuously on all X workstations) to the SAFstation client process. The SAFstation client process probes the X windows event queue frequently. It ticks faster than any other part of the ModSAF system (this is required because X uses a point-to-point reliable communications service, which if not properly drained, can completely seize the workstation). The ModSAF scheduler (libSched) invokes a function which pulls the button press event off the event queue and passes it to the Xt Intrinsics library. See section "Overview" in *LibSAFGUI Programmer's Reference Manual.*
See section "Overview" in *LibSelect Programmer's Reference Manual.*
See section "Overview" in *LibSched Programmer's Reference Manual.*

6.1.1.2 Select Mode to Create Mode

The Xt Intrinsics library dispatches the button press event to libSAFGUI in the form of a callback. LibSAFGUI responds to this event by first confirming that the privilege level of the workstation is set to allow creation of vehicles (using libPrivilege) and if so, changes the *mode* of the workstation to vehicle creation mode. Because the user interface is *modal*, it can only exist in one mode. Activating a new mode deactivates the previous one. Triggering a new mode can either be a *push* or a *replace* operation. For example, if the workstation is in the base or select mode, libSAFGUI assesses the transition from select to creation modes using internal rules and determines that a *push* operation be used.

A mode *push* is accomplished by first suspending the current mode, then starting the new mode. LibSAFGUI issues a callback to suspend the select mode editor, libSelect. LibSelect attaches callbacks to the vehicles and graphics on the map (using a service called libSensitive). These callbacks would normally cause libSelect to start an editor to modify or inspect the object selected. However, the select mode is suspended, so libSelect must remove the callbacks that it has attached to those items. In addition, it tells libSensitive (the mouse sensitivity service) to desensitize all the icons (so they do not appear *hot* or highlighted). Finally, libSelect removes the buttons (if any) it has placed in the editor tile of the interface. See section "Overview" in *LibCallback Programmer's Reference Manual*.
See section "Overview" in *LibSensitive Programmer's Reference Manual*.

6.1.1.3 Unit Creation Editor Display

The select mode suspend callback completes, and the thread of execution returns to libSAFGUI. LibSAFGUI issues another callback to the service registered to handle the create mode, libUnits. LibUnits passes this callback through to libEditor which posts the editor window that libUnits created (from a data file) at program startup. LibEditor calls the Xt Intrinsics Library to display the unit creation editor. It initializes all the editor fields from a set of initial values specified in the data file for this editor. One of these fields (in this case, the location) is specified as a *Forced* value. (You must provide the value.) LibEditor calls libSAFGUI to display a special help message, indicating that a forced choice exists. The mouse input focus is directed to the forced choice editable. See section "Overview" in *LibUnits Programmer's Reference Manual*.
See section "Overview" in *LibEditor Programmer's Reference Manual*.

6.1.1.4 Return Control to the Scheduler

At this time, libSAFGUI finishes its job (switching from select mode to create mode) and returns from the X event handler that started this chain of events. Many other X events may be queued since the management (X terminology for whether a piece of the interface is displayed) of many parts of the user interface has changed. Processing X events continues until there are none left. Control is returned to libSched.

6.1.2 Placing the Unit

The following describes the events that occur while the SAFstation waits for you to input the vehicle's location, the location is converted to a coordinate system, the unit symbol is displayed, and the current mode is exited.

6.1.2.1 Input a Location

The SAFstation is again in a quiescent state. LibPktValve is polling the network, X is being polled for incoming information, and libSched is periodically ticking various other services. Since there is little activity at this point, the system is in an idle loop in the scheduler. When the scheduler is idle, it spins on a read of the real time clock and waits for the next event to be ready for execution.

The unit creation editor has fields corresponding to many attributes of ModSAF units, including unit type, location, direction, marking, etc. You can edit any of these parameters in this editor window. If you find the default values acceptable, (except for the location of the unit – as a forced choice, the location has no default value), you need to then click on the tactical map to indicate a location.

This mouse event is handled by libSensitive which determines that it was a map click (in contrast to a click on a sensitive object such as a point or a line). It fires a callback event announcing the click to the higher layer services that have subscribed to it.

6.1.2.2 Location to a Coordinate System Conversion

When the focus is on the PLACE editable (a Motif text-editing widget that accepts locations), a callback is registered with libSensitive specifying an interest in map events. This callback is executed in response to the unit placement mouse click. The callback for map clicks in libEditor is directed at the PLACE editable and looks at the screen coordinates of the mouse click. Using the tactical map service (libTactMap), the coordinates are translated to the internal coordinate system of ModSAF (TCC or TopoCentric Coordinate System). It converts the resulting "X" and "Y" coordinates to the preferred system using the coordinate conversion service (libCoordinates). LibEditor also translates the coordinates to text strings (displayed on the text editing widgets comprising the PLACE editable) so that you can edit them at a later time. In response to the text strings being set, the Motif text-editing widgets trigger *valueChanged* callbacks in libEditor. These callbacks parse the new text values and store them internally as the value of the PLACE editable. The *valueChanged* callbacks allow all interpretation of values and handling of input to pass through a single set of callback routines. Thus, the same callbacks are triggered if you modify the value of the text fields directly in the PLACE editable.

See section "Overview" in *LibTactMap Programmer's Reference Manual.*
See section "Overview" in *LibCoordinates Programmer's Reference Manual.*

6.1.2.3 Unit Symbol Display

The unit symbol is created at initialization by the unit creation editor using libTactMap. If the internal values of the PLACE editable are updated, they pass back to the higher layer service (in this case, libUnits) that created the editor via a *render* callback. The editor's configuration file drives which editables trigger render callbacks. LibUnits responds to the render callback by updating the object on the tactical map. If there are any changes in the location and/or specific unit type, the tactical map service is notified. The tactical map service erases any previous version of the unit symbol and redraws it at the newly specified location.

If you drag the mouse over the screen, this entire sequence is repeated for every mouse motion event, resulting in the apparent dragging of the unit symbol.

See section "Overview" in *LibUnits Programmer's Reference Manual.*

6.1.2.4 Exiting the Current Mode

After the location is specified, libEditor checks if a forced choice has been resolved. In this case, it has, so the editor makes the `Done` button sensitive (if forced choices are outstanding, the user is only allowed to abort).

You find the default values acceptable and press the `Done` button. In response to the `Done` event, libEditor sets an internal flag indicating a normal exit status (in contrast to an `Abort` exit) and asks libSAFGUI to exit the current mode.

6.1.3 Creating Persistent Objects

The following is a description of the events that occur when the editor is closed, Persistent Object (PO) prototypes are created, and Protocol Data Units (PDU's) are transmitted.

6.1.3.1 Closing the Editor

LibSAFGUI performs a sequence of actions similar to that which occurred when the create mode was pushed. Among them is notifying libUnits, via a callback, that it no longer is the current mode. LibUnits, which manages the editing of unit class persistent objects, passes this message through to libEditor.

LibEditor removes its interface widgets from the editor tile and then sends a callback to libUnits to notify it that the editor has finished. LibEditor passes the exit status: "Done" or "Abort." If "Done," then libEditor passes the object data (needed later to create a PO).

6.1.3.2 Creating a PO Prototype

LibUnits takes the object data and calls libUnitUtil to create the necessary objects in the PO database. An object is added to the PO database by creating a prototype in a local variable, and then passing this prototype for creation to libPO, which provides an interface to the shared PO database.

The objects in the database created by libUnitUtil represent 1) the units in the echelon created, 2) the overlays used by those units for planning and 3) the background task frames that hold all the unit's innate behaviors. Thus, a platoon of four tanks is initially represented by 15 objects (unit, overlay, and background frame [3], for the platoon, and its four vehicles [times 5]). LibUnitUtil finds the composition of the unit using the echelon database service, libEchelonDB. See section "Overview" in *LibPO Programmer's Reference Manual.*
See section "Overview" in *LibUnitUtil Programmer's Reference Manual.*
See section "Overview" in *LibEchelonDB Programmer's Reference Manual.*

6.1.3.3 Transmitting PO PDUs

LibPO checks the prototype for errors or *invalid* content. If there are no errors, libPO creates a local representation of the object in a structure, called a `PO_DB_ENTRY`. A single `PO_DB_ENTRY` describes the state of an object in the shared PO database at one time. LibPO then places a Describe Object PDU on the network announcing the existence of this new object. The local copy is *threaded* into several lists of objects within the PO database, including a retransmission queue, that is used for periodic retransmission of the Describe Object PDU.

The unit object that represents the platoon contains a flag called `shouldBeSimulated` (set by libPO) that indicates a SAFsim should create a platoon.

6.1.4 Creating C2 Class Objects

Every SAFsim on the network receives the PDUs sent by libPO describing all new objects. For each new object that appears in the PO database, a *new object* callback fires. The same callback fires regardless of whether the object was created within this process or a packet was received over the network. This uniformity allows the SAFsim and SAFstation processes to reside within the same executable. The two processes act independently on the same callbacks.

For each new object in the PO database, a corresponding C2 class object is created. This is where local information about the object resides. For example, within a SAFstation every point, line, and area class object in the PO database is represented on the tactical map by an object created by the tactical map service. The handle to this object is stored as part of the *graphic* C2 subclass information. The graphic subclass attached to the units just created appear on the user interface (if their display is enabled) through libTactMap.

To follow the flow of data into a single SAFsim process, the ModSAF scheduler is needed. One of the periodic functions that is posted on the scheduler reads the network. This function is provided by the packet valve service, libPktValve. When it is called, it extracts all the new packets, filters those that are not of interest, and fires an event for each packet. The packets are contained in memory provided by the queuing service, libQueue. This memory (*queue buffer*) is tagged with a reference count so that many different recipients of the same queue buffer can save it for later processing. Each deallocates it individually, and the memory is freed when the number of references reaches zero. The ModSAF main program subscribes to incoming packets in the PO protocol family. The handler sends the packets to libPO for processing. When libPO receives the packets, it discovers that a new object is being described and fires the *new object* callback event.

One recipient of the new object callback is libC2obj, the C2 object class library. It responds by creating an instance of a C2 class object. This involves a call to the class service (libClass) to allocate a block of user data, a call to libPO to attach this point to the `PO_DB_ENTRY` in the PO database, and then a sequence of calls to all the C2 object subclasses, to allocate and attach their own instance data structures. In this case, C2 class objects are created for all the new units, task frames, and overlays.

After the platoon, its subordinates, and related objects are created, the SAFstation process modifies the platoon object to set the `shouldBeSimulated` flag. This modification goes through the PO database service again. A packet is transmitted on the network resulting in *object changed* callbacks firing on all SAFstations and SAFsims.

6.1.4.1 Simulation Nomination

The vehicle creation library, libCreate, receives the *object changed* callback. LibCreate notices the setting of the `shouldBeSimulated` flag and starts a negotiation process for the creation of the unit. All SAFsims on the network use the same steps to determine which simulator is the best candidate for simulating this particular unit. Changes are made to the platoon object in the PO database by *every* SAFsim indicating which SAFsim should perform the simulation.

The conflict resolving properties of the PO database result in a uniform decision of one SAFsim to perform the simulation. All SAFsims request that libSched call them back after a short delay.

When this deferred function is invoked by the scheduler on each SAFsim process, each inspects the current state of the platoon object in the PO database. Only one SAFsim is selected by the group to be responsible for simulating the unit.

See section "Overview" in *LibPktValve Programmer's Reference Manual.*
See section "Overview" in *LibQueue Programmer's Reference Manual.*
See section "Overview" in *LibC2Obj Programmer's Reference Manual.*
See section "Overview" in *LibClass Programmer's Reference Manual.*
See section "Overview" in *LibCreate Programmer's Reference Manual.*

6.1.5 Creating SAF Class Objects

The following is a description of the events that occur when SAF class objects are created.

The SAFsim selected to create the platoon iterates over each subordinate (the tank objects). It finds these subordinates by *querying* the PO database (a search through all the objects). For each tank that is to be simulated, it calls libDISDBObj, to create a local vehicle.

Like libC2obj, libDISDBObj uses libClass to allocate the block of user data for each vehicle. It assigns the vehicle a unique identifier, and creates an entry in the global vehicle table (created at startup time by libVTab and is the base of the DIS database abstraction). LibDISDBObj attaches the allocated user data to the entry in the vehicle table.

LibDISDBObj calls libParMgr as a service to create each DIS DB object subclass. Each DIS DB object subclass is assigned a unique SAF Model number for identification. The SAF Model number is read in from the protocol header files and registered with the initialization routine of each library. LibParMgr consults its database of vehicle types to see if the SAF Model should be included in the passed vehicle type. If so, that model's data file parameter parsing routine is called to translate the parameters in the data file into an internal structure. If the model does belong in vehicles of the passed type (for example, if a radio belongs in an M1 tank), LibParMgr passes this data to the individual subclass library's create routine.

The create routines for the subclass libraries allocate local storage to hold information about the entity, can create other objects in the PO database, post their own callback event handlers, register tick handlers with libDISDBObj for the subclass, and so on. For example, the innate behaviors represented by *background tasks* create tasks in the PO database and place them in the background frames created earlier.

After letting LibParMgr perform the "lookup-then-create" action for every possible subclass, an entire SAF entity has been constructed. LibDISDBObj requests that the scheduling service call the `tick` routine for this vehicle periodically. The key to the new entity, called its `vehicle_id`, is returned to libCreate.

LibCreate takes all the initialization information implicit in each unit (location, direction, etc.), and calls DIS DB object subclasses (such as libEntity) to set the initial state.

See section “Overview” in *LibDISDBObj Programmer's Reference Manual.*
See section “Overview” in *LibVTab Programmer's Reference Manual.*
See section “Overview” in *LibParMgr Programmer's Reference Manual.*
See section “Overview” in *LibEntity Programmer's Reference Manual.*

6.1.6 The First Tick

The following is a description of the events that occur during the entity's first tick as DIS PDU's are transmitted, remote vehicles are updated, and the platoon appears on the tactical map on the GUI.

6.1.6.1 Transmitting DIS PDUs

After the scheduler reaches the libDISDBObj tick function for one of these tanks, it invokes this function which, in turn, invokes the registered tick functions for the DIS DB object subclasses of the vehicle. One of these is libEntity's `ent_tick` routine. This routine notices that it has never sent an Entity State DIS PDU on the network and invokes a routine to send the PDU. A block of memory for the PDU is allocated using libQueue, and the packet is built in this memory. Local entity state information such as location, velocity, and entity type are translated as needed to the formats on the network. LibEntity then calls libPktValve to transmit the PDU.

LibPktValve sends the packet using calls to the operating system, and checks whether the packet should be looped back for further processing within the ModSAF system.

This PDU is received by each SAFstation and SAFsim process on the network, and is dispatched by the packet valve services in those processes to all interested modules.

6.1.6.2 Remote Entity Creation

LibRemote receives the Entity State PDU over the network and updates the state of the internal DIS database with the current state. Handling a new remote entity state is similar to handling a new local; libDISDBObj is called to create the entity in the DIS database (using libVTab, libClass, etc.). The subclasses that apply to remote vehicles are created. Note: The set of subclasses for local vehicles is defined in a data file. However, the subclasses that apply to remote vehicles is fixed because remote entities are updated from the network and the available information is fixed by the DIS protocol. The fixed set includes entity, position-based table management, and user interface subclasses. Each subclass decides whether it should be included as part of a remote vehicle by posting handlers with special create and destroy events defined in libDISDBObj.

See section "Overview" in *LibRemote Programmer's Reference Manual.*

6.1.6.3 Platoon Display

LibPVD displays the platoon unit on the Plan View Display. The PVD subclass creates a tactical map object to represent the vehicle graphically (libBGRDB provides the service for drawing military graphics). It uses the formation database service, libFormationDB, to find the correct positions for each vehicle in the platoon. This graphic is periodically updated from the latest information in libEntity.

The creation process is complete.

See section "Overview" in *LibPVD Programmer's Reference Manual.*
See section "Overview" in *LibBGRDB Programmer's Reference Manual.*
See section "Overview" in *LibFormationDB Programmer's Reference Manual.*

7 Extending ModSAF

ModSAF's open architecture facilitates the ability for developers from many different organizations to add new vehicle types, echelons, weapons, and behaviors. This chapter covers the steps involved in making these extensions.

Note: Throughout this discussion, "you" is referred to as the developer. The DIS environment imposes some limits on the type of extensions that can be made. The primary one is the limitation of using standard protocol constants. The SIMNET and DIS protocols both use a taxonomy of object type codes to describe entities and weapons. For example, there is a constant in SIMNET called `vehicle_US_M1` that represents an M1 tank. There are no separate constants for the M1-A1, M1-A2, and so on. Adding these constants would be prohibitively costly, since it would require modifying all existing SIMNET simulators. However, since the object types are defined as a taxonomy, it is possible to write simulations that tolerate unknown object types by using a defaulting scheme. ModSAF is such a simulation, and thus ModSAF systems can interoperate reasonably when confronted with some unknown object types.

Note: If you find that the existing protocols cannot adequately describe the object type that is going to be added, you should contact Loral (see Section 1.1 [ModSAF Q&A], page 1) to make the necessary changes to the standard protocols.

For experiments in which interoperability with other DIS or SIMNET simulators is not a requirement, and for initial developmental purposes, small changes to the protocol can be made to the following files:

- '`obj_type.h`', '`veh_type.h`', '`mun_type.h`', '`unit_type.h`', and '`p_safmodels.h`', in the directory '`common/include/protocol`', all contain many protocol constants. To include a new constant, modify the SIMNET or PO protocol definition in the appropriate file.
- '`disconst.rdr`' contains the source data for SIMNET to DIS translations. Modify this file in libdisconst, if needed, and do a '`gmake`' in this directory. See section "Overview" in *LibDISConst Programmer's Reference Manual.*
- In order to allow these new constants to be read from the data files, a '`gmake`' must be executed in the directory '`common/libsrc/librdrconst`'.

  See section "Overview" in *LibRdrConst Programmer's Reference Manual.*

How to implement these changes while extending ModSAF is described in more detail in the following sections.

7.1 Adding Vehicles

The following sections describe the necessary modifications to the files responsible for adding a vehicle. The procedure can be broken down into the following steps.

1. Define the protocol constants of the new vehicle
2. Create or modify the vehicle parameter file
3. Load the parameter file
4. Reference the new parameters
5. Add the vehicle to the graphical user interface
6. Add vehicle and military icons
7. Check the physical database

Each step is explained in more detail in the following sections.

7.1.1 Define the Vehicle

A protocol constant must be defined for the new vehicle. For example, to create a new USM1A3, you must add the following entry to '`common/include/protocol/veh_type.h`':

```
        /* M1A3 main battle tank:  */
    #define SP_vehicle_US_M1A3              \
           ( SP_objectDomainVehicle | SP_vehicleEnvironmentGround \
            | SP_vehicleClassSPArmoredTracked | SP_vehicleCountryUS \
            | 1 << SP_vehicleSeriesShift | 3 << SP_vehicleModelShift \
            | SP_vehicleFunctionMainBattleTank )
```

When possible, insert the new vehicle along with others of its kind, and modify the series and model shift appropriately. Most importantly, when defining a new vehicle, always have every definition be *unique*.

The specific object domains, environments, vehicle types, country codes, and function are listed in '`/src/protocol/obj_type.h`'. Although new definitons can be added to obj_type.h, a modification is rarely necessary since this protocol file is extensive.

After completing this modification, do a '`gmake`' in '`libsrc/librdrconst`' to copy the new constant to the data files. For more information, See section "Overview" in *LibRdrConst Programmer's Reference Manual.*

You will also want to be familiar with the file '`libsrc/libdisconst/disconst.rdr`', where SIMNET to DIS translation tables reside. In general, the vehicle you are adding is already defined. However, if 1) you are adding a new vehicle that does not exist and 2) you are going to simulate across the network, 3) you need to make modifications to '`disconst.rdr`' and, 4) subsequently do a '`gmake`' in '`libdisconst`' to install the changes. For a detailed description of the format of the translations,

See section "Overview" in *LibDISConst Programmer's Reference Manual.*

7.1.2 Create the Parameter File

To define the model parameters of the new vehicle, a new parameter file must be created. This is most easily done by copying and modifying an existing parameter file. Locate an existing vehicle that has characteristics most closely resembling the vehicle you are creating. Then, edit the file of the similar vehicle that you copied. For example, to create the M1A3 copy the parameter file of the M1A2 '`src/ModSAF/entities/US_M1A2_params.rdr`' to '`US_M1A3_params.rdr`', and replace all instances of US_M1A2 with US_M1A3, as shown below.

```
US_M1A3_MODEL_PARAMETERS

(SM_Entity DEFAULT_DEAD_RECKONING_PARAMETERS
           (vehicle_class vehicleClassTank)
           (guises vehicle_US_M1A3 vehicle_USSR_T72M)
           (send_dis_deactivate true)
           )
```

Modify the model parameters of the M1A3 accordingly. Note: The parameters of each subclass (each `SM_Lib` in the data file) are documented in the texinfo documentation of the library that implements that subclass. For instance, the `SM_Entity` parameters are documented in libEntity.

See section "Overview" in *LibEntity Programmer's Reference Manual.*

7.1.3 Load the Parameter File

The file, '`src/ModSAF/entities/modellist.rdr`', lists the parameter files that are loaded at program startup. The name of the new parameter file must be added to this list. Since order is *very* important, add the new file just prior to the comment `;; Add more vehicles here`:

```
 "US_M1A3_params.rdr"
;; Add more vehicles here
```

7.1.4 Reference the Parameter Macro

The data files for the individual vehicles are macro definitions. These macros are referenced in the file '`src/ModSAF/entities/models.rdr`'. This file contains a list of all vehicles, lifeforms, and munitions. It must be modified to make the association between the vehicle type name and the model parameters for that vehicle:

```
("vehicle_US_M1A3"       US_M1A3_MODEL_PARAMETERS GROUND_STD_PARAMS)
```

7.1.5 Add to the User Interface

The file '`/libsrc/libunits/units.rdr`' specifies the list of vehicle and unit types that can be created from the Unit Editor. You must add the new vehicle to this list.

```
("M1A3"          vehicle_US_M1A3)
```

Execute the command '`gmake`' in this directory to install the modification.

See section "Overview" in *LibUnits Programmer's Reference Manual.*

7.1.6 Add the Icons

The object type defaulting scheme applies to the new vehicle. Most likely, an acceptable vehicle icon will appear on the PVD display. However, if it is necessary to specify a different icon, edit the file '`libsrc/libpvd/pvd.rdr`'. This file defines pictures and maps them to vehicles. They are scaled to size based on the values in '`physdb.rdr`'.

See section "Overview" in *LibPVD Programmer's Reference Manual.*

You can specify an entry in '`libsrc/libbgrdb/bgrdb.rdr`' that relates object types to military icons displayed for that object (platform). You can also specify whether the icon rotates as the platform rotates.

See section "Overview" in *LibBGRDB Programmer's Reference Manual.*

7.1.7 Check the Physical Database

Information about the physical attributes of an entity are stored in the physical database file '`common/libsrc/libphysdb/physdb.rdr`'. This database uses the object type defaulting scheme if necessary, but should be updated with specific information about every new vehicle. The format of this data file is explained in comments at the top of the '`physdb.rdr`' data file.

See section "Overview" in *LibPhysDB Programmer's Reference Manual.*

7.2 Adding Echelons

To add a new echelon (or new formation of units), follow these steps:

1. Define new protocol constants for the new echelon
2. Add the echelon to the graphical user interface
3. Create the structure of the echelon

Each step is explained in more detail in the following sections.

7.2.1 Define the Echelon

Ensure that a unique protocol constant exists to describe the new unit. DIS currently does not have protocol constants for echelons, so the only data file involved is '`common/include/protocol/unit_type.h`'. As with any other protocol modification, notify Loral (see Section 1.1 [ModSAF Q&A], page 1) to coordinate any changes.

7.2.2 Add Echelon to the User Interface

The file '`common/libsrc/libunits/units.rdr`' specifies the list of unit types that can be created from the Unit Editor. Add the new echelon to this list. Execute the command '`gmake`' in this directory to install the modification.

See section "Overview" in *LibUnits Programmer's Reference Manual.*

7.2.3 Create the Echelon Structure

Add a definition of the structure of the new echelon to the echelon database file, '`common/libsrc/libechelondb/echelondb.rdr`'. For a description of how to define an echelon,

See section "Overview" in *LibEchelonDB Programmer's Reference Manual.*

7.3 Adding DIS DB Object Subclasses

The following sections describe the necessary modifications to the files responsible for adding a new vehicle subclass. The procedure can be broken down into the following steps:

1. Define protocol constants for the new subclass
2. Create or modify a vehicle class library
3. Initialize the library
4. Add necessary calls and rebuild
5. Modify the Makefile
6. Modify the modsaf.libs file

Each step is explained in more detail in the following sections:

7.3.1 Define the Subclass

To define a protocol constant for the new DIS DB object subclass, modify the file '`p_safmodels.h`' in the directory '`common/include/protocol`'. Since this is a protocol file, it is critical that additions be coordinated between organizations. Contact Loral (see Section 1.1 [ModSAF Q&A], page 1) with a description of the new subclass, and a SAF model number will be issued.

7.3.2 Create the Library

To add a new vehicle subclass, a library must be created. A program calleld '`osatemplate`' is included in the ModSAF distribution to simplify this process. The program is used as follows:

`% cd common/libsrc`
: The templating program expects to be run in this directory.

`% osatemplate`
: Start the templating program. It resides in '`common/bin/<arch>`'.

`module name (e.g., entity, pbtab)?`
: This is the name of the library. Choose a name that clearly explains what the library will implement, and does not conflict with any other libraries.

`file prefix (e.g., ent, pbt)?`
: This specifies the short sequence of characters that will start all files in the library. It should be unique, and *must not exceed 6 characters.*

`function prefix (e.g., ent, pbt)?`
: This specifies the prefix that will start each function in the library. Usually, this will be the same as the module name.

`SAF model number (e.g., SM_Entity, SM_PBTab)?`
: This specifies a protocol constant from the file '`common/include/protocol/p_safmodels.h`'.

```
Built ./lib<module>. You should now customize the library.
Start by doing a grep for 'TEMPLATE' in all the files.
```
: As the message states, the library will be constructed. This is a fully ModSAF compliant library, including all necessary source files and documentation to be used in the ModSAF program. Of course, it doesn't do anything yet. All the places where changes are needed are marked with the word `TEMPLATE`.

Instead of using '`osatemplate`', it is often easier to find a similar library, copy it, and make the appropriate modifications.

7.3.3 Initialize the library

Incorporate the new library (lib<module>) into the ModSAF program structure as follows:

1. Include the new header file in '`common/src/ModSAF/main.c`':

   ```
   #include <lib<module>.h>
   ```

2. Add a line to initialize the library globally (still in '`main.c`', in the neighborhood of similar initialization calls):

   ```
   /* simulation of a <module> */
   <module>_init();
   ```

Note: When the subclass is a *components* subclass, creation is implied via the registration process in class initialization, (see section "cmpnt_define_instance" in *LibComponents Programmer's Reference Manual Index*). If you are creating a components subclass, register the subclass create and destroy routines with libcomponents as opposed to libparmgr.

7.3.4 Add Necessary Calls and Rebuild

1. Any existing libraries that are modified to refer to a new SAF model number, or call any new object functions must be rebuilt with '`gmake`' after the new object is built.
2. Rebuild '`libsrc/librdrconst/constants.rdr`', (with '`gmake`'), so that the new SAF model constant is recognized by libRdrConst.

7.3.5 Modify the Makefile

The following describes the steps to take when modifying the Makefile:

1. The Makefile in '`common/src/ModSAF`' must be modified to include your new library. However, the new library must be inserted in the correct order, so that the libraries that depend on it are listed before the new library. To find the correct place for this new library, you can either figure out the dependencies yourself and insert the new library name accordingly, OR, use the '`depends`' program in the directory '`libsrc`'. If you use the '`depends`' program, the following commands create an output file that lists all the libraries (including the new library, lib<module>) in the correct order for linking:

   ```
   % cd common/libsrc
   % depends -l <outfile> lib<module> `cat modsaf.libs`
   ```

Note: The flag for the `<outfile>` option is the letter l and not the number 1. Also, be sure to use ‘ and not ’ (for the argument ‘cat modsaf.libs‘).

If you do not have the ‘`modsaf.libs`’ file in ‘`common/libsrc`’, copy it from ‘`common/src/ModSAF`’. Normally, ‘`modsaf.libs`’ is copied automatically when you are building the entire ModSAF repository with ‘`make-all`’.

Note: To find out forward dependencies (i.e. what libraries lib<module> depends on), type:

```
% depends -c lib<module> ‘cat modsaf.libs‘
```

Note: To find out reverse dependencies (i.e. what libraries depend on lib<module>, type:

```
% depends -r lib<module> ‘cat modsaf.libs‘
```

2. An ascii file <outfile> is generated by the ‘`depends`’ program. Insert <outfile> into ‘`common/src/ModSAF/Makefile`’, replacing the outdated linker information.
3. Rebuild ModSAF in the directory ‘`common/src/ModSAF`’ by doing a ‘`gmake clean`’, followed by a ‘`gmake`’.
4. The new linker information generated by ‘`depends`’ includes links only for ModSAF compliant libraries. Thus, do *not* delete from the Makefile the links to the non-compliant libraries, which are libparser, libtty, libp2p, libassoc, libnetif, libmove, and libbgr. These links are at the end of the linker information in the ModSAF Makefile.

7.3.6 Modify modsaf.libs

The ‘`modsaf.libs`’ file contains an alphabetical list of all libraries on which ModSAF depends. This file is copied from the directory ‘`common/src/ModSAF`’ to the directory ‘`common/libsrc`’ when executing a ‘`make-all`’. These files should be kept current with each library addition so that the program ‘`depends`’ can generate an accurate, comprehensive list when it uses ‘`modsaf.libs`’ to extract inter-library dependencies. Therefore, when you have completed writing the task and finally know the dependencies, you should add the new library name to both copies of ‘`modsaf.libs`’.

7.4 Adding Weapons

New weapons systems can be constructed by creating sets of parameters for use by the existing weapons simulations (libBalGun, libMLauncher, libMissile, etc.). In general, you must perform the following steps:

1. Check the physical database to be sure the necessary turret, gun, missile launcher, or other component is represented.

   See section “Overview” in *LibPhysDB Programmer's Reference Manual.*

2. Add the specific information regarding the new weapon to the data file for the platforms on which it will be mounted. These data files are in the directory '`common/src/ModSAF/entities`'. An easy way to accomplish this is to copy the data file of a similar weapon system on a similar vehicle. For details about the meaning of the various parameters, see the library documentation for the model. For example, for an explanation of the parameters of ballistic guns,

   See section "Overview" in *LibBalGun Programmer's Reference Manual.*

3. Provide munitions for the weapon by modifying the `SM_Supplies` parameters in the data file for the vehicle.

   See section "Overview" in *LibSupplies Programmer's Reference Manual.*

4. Include the new gun or launcher as a component by modifying the `SM_Components` parameters in the data file for the vehicle. Be sure to give the new component a unique instance number (the small number that is attached to the SAF model number with an `|` in the data file). Also check the documentation of the library to ensure that the maximum number of instances of each component is not exceeded.

   See section "Overview" in *LibComponents Programmer's Reference Manual.*

5. Add the weapon to the parameters `SM_VAssess` that indicate the types of targets on which it is effective.

   See section "Overview" in *LibVAssess Programmer's Reference Manual.*

6. Modify the parameters of the `SM_VTargeter` with details of how the weapon is used.

   See section "Overview" in *LibVTargeter Programmer's Reference Manual.*

7. If the munition fired by the weapon is a missile, and that missile is not already defined, then the steps detailed for adding new vehicle types (See Section 7.1 [Adding Vehicles], page 61, for more info) must be followed for the missile as well.

7.5 Adding Behaviors

New behaviors are added to the ModSAF system in the form of tasks. It is critically important that before adding a new task, you review all of the following documents:

See section "Overview" in *LibTask Programmer's Reference Manual.*
See section "Overview" in *LibTaskMgr Programmer's Reference Manual.*
See section "Overview" in *LibTaskEdit Programmer's Reference Manual.*
See section "Overview" in *LibTaskPri Programmer's Reference Manual.*

7.5.1 Create a DIS DB Subclass

Task libraries are added using the same steps outlined previously for adding DIS DB object subclasses. Follow the guidelines presented in See Section 7.3 [Adding DIS DB Object Subclasses], page 65, namely

1. Define protocol constants for the new subclass
2. Create a task library
3. Initialize the library
4. Add necessary calls and rebuild
5. Modify the Makefile
6. Modify the modsaf.libs file

The only difference is that instead of using the program '`osatemplate`', run '`tasktemplate`' in '`common/libsrc`' to create a task library.

Note: Sometimes, it is easier to find a task that performs a similar function and copy it to the new library name. However, be sure to make the necessary modifications.

7.5.2 The Task Library

The program '`tasktemplate`' creates a fully ModSAF compliant task library, equipped with all the files and interfaces necessary for implementing a new behavior. This section briefly describes the various parts of the task library and refers the reader to the relevant *Programmer's Reference Manual.*

7.5.2.1 Makefile

The Makefile for task libraries is different from other libraries, because the `.fsm` file (where the task behavior is defined) leads to generated source files.

7.5.2.2 Finite State Machine

- Task behavior is defined in `<fn_pref>_task.fsm`. Begin by including the necessary header files. Following the include files is an opening comment that identifies the variables available within the state machine code fragments in addition to the events defined in tasks (i.e. tick, params).

See section "Opening Comment" in *LibTask Programmer's Reference Manual.*

- The opening statement for the finite state machine is also automatically created by the task template program. This is where task parameters, task state, and private task variables are specified.

  See section "Name and Structure Declarations" in *LibTask Programmer's Reference Manual.*
- Tasks may optionally initialize tasks and define frames in which to put them. These frames are referred to in the '`.fsm`' file as `SUBFRAMES`. The tasks that compose the `SUBFRAMES` are referred to as `SUBTASKS`. See section "Sub-task Declarations" in *LibTask Programmer's Reference Manual.*
  See section "Sub-frame Declarations" in *LibTask Programmer's Reference Manual.*
- You can declare the criteria under which you want your task to take control of a given set of actuators. Priorities are defined in the `SM_TaskPriority` subclass in '`common/src/ModSAF/entities/standard_params.rdr`'. LibTaskPri ultimately makes the decision as to which task controls which critical resources.

  See section "Criteria Declarations" in *LibTask Programmer's Reference Manual.*
  See section "Overview" in *LibTaskPri Programmer's Reference Manual.*
- At the minimum, tasks begin with a standard set of events, i.e. `tick()` and `params()`. Additionally, custom events can also be defined.

  See section "Event Declarations" in *LibTask Programmer's Reference Manual.*
- Following the event definitions is the START body, that starts the task, initializes variables, and ensures acceptable task transitions from one state to another. The END body executes when a task is ended and the optional SUSPEND body executes if a task is suspended. See section "Start Body" in *LibTask Programmer's Reference Manual.*
  See section "State Definitions" in *LibTask Programmer's Reference Manual.*
  See section "End Body" in *LibTask Programmer's Reference Manual.*
  See section "Suspend Body" in *LibTask Programmer's Reference Manual.*
- A set of special commands facilitates task writing with the help of the AAFSM Code Generator. See section "Special Commands" in *LibTask Programmer's Reference Manual.*
- The C functions used in the state machine or that are public are defined at the end of the '`.fsm`' file.

7.5.2.3 Public Header File

- Add the model parametric data or system parameters to the public header file, '`lib<module>.h`'. The structure is called `<module>_PARAMETRIC_DATA`.

- These system parameters are shared and public, are specific to the entity type, and can be used to specialize a generic task. Default system parameters are specified in `'src/ModSAF/entities/standard_params.rdr'`.
- Add the model parameters or task parameters to the public header file. These parameters are shared and public, too. The task parameters structure, `<module>_PARAMETERS`, stores values that are inputs to the task when it is spawned.
- The macro `<MODULE>_MAX_TASK_REFERENCES` specifies the number of objects (of type `ObjectID`) referred to in the task parameters. All `ObjectID` type parameters must be listed before any other type in the task and system parameters structures.
- Default task parameter values *must* be specified in `'<module>.rdr'`. The parameters in this reader file, as well as in `'common/src/ModSAF/taskframes.rdr'`, build the task editor in the user interface. More information about these reader files follows later in these guidelines.

  See section "Overview" in *LibParMgr Programmer's Reference Manual.*
  See section "Overview" in *LibParmEdit Programmer's Reference Manual.*
  See section "Overview" in *LibTaskEdit Programmer's Reference Manual.*

7.5.2.4 Local Header File

- Add the state variables or shared task state to the local header file, `'lib<file_pref>_local.h'`. The structure is defined as `<module>_STATE`.
- The shared task state is shared and private, can be considered as outputs from a task, and should not change frequently.
- Add the local task state parameters to the structure `<module>_VARS`. The macro `<MODULE>_STATE_REFERENCES` specifies the number of objects referred to in the state parameters
- The local task state is local and private, can change frequently, and usually has a pointer to the system parameters.

7.5.2.5 Initialization

- The file `'<file_pref>_init.c'` contains the function that initializes the task (this function is called in main.c).
- The first function is usually the **`status`** function. This function is registered with the status monitor through the function call `statmon_register` in `<fn_pref>_init`. The **`status`** function provides messages to you at run-time describing the status of the vehicle tasks (Note: the status monitor is displayed on the GUI in the Unit Operations Editor). Generally, the

`status` function contains one `switch` statement with one `case` for each possible state. Strings are formatted for output to the GUI via `sprintf's` or `edt_format` (when the output must conform to the User's Preferences).

See section "Overview" in *LibEditor Programmer's Reference Manual.*
See section "Overview" in *LibStatMon Programmer's Reference Manual.*

- The file, '`<fn_pref>_init`' also contains the function that registers the predicates and the Before and After lists (i.e. lists of tasks from which to receive input, and for which to provide output, respectively). This function initializes the params field, registers the editor, registers any protocol conversions, and initializes the status monitor.

7.5.2.6 Utility Functions

- The '`<file_pref>_util.c`' file holds the state initializing function. The function, `<fn_pref>_init_task_state`, is called whenever the task is spawned (either from the editor or from another task). This function is used to initialize the `size`, `refcount`, and `data` fields of the `state` variable. For more information on the `init_fcn` parameter of `taskedit_register`, See section "Overview" in *LibTaskEdit Programmer's Reference Manual.*

7.5.2.7 Class Information

- The '`<file_pref>_class.c`' file manages class information attached to each vehicle. The function, `<fn_pref_print>`, may be modified to print out values of the private variables or the `VARS` data structure. The `<fn_pref_create>` and `<fn_pref_destroy>` functions are also defined in this file.

7.5.2.8 Parametric Data

- The '`<file_pref>_params.c>`' file manages the parametric data. The parametric data are the system parameters (`<module>_PARAMETRIC_DATA`); their default values are stored in '`src/ModSAF/entitie`
- The '`<file_pref>_params.c`' file sets up the `parser` and `changer` functions to be registered with LibParMgr in `<fn_pref>_init_params`. The `parser` function reads in the system parameter values from '`standard_params.rdr`' using LibReader. The `changer` function is called whenever the task parameters are changed (e.g. through a task editor).

  See section "Overview" in *LibParMgr Programmer's Reference Manual.*
  See section "Overview" in *LibReader Programmer's Reference Manual.*

7.5.2.9 Reader File

- The '`<module.rdr>`' file provides information to the task editor, libEditor. This file contains a structure for the model parameters. Again, these are the task parameters (`<module>_PARAMETERS`), as defined in the public header file. Each of these fields *should* be initialized in '`common/src/ModSAF/taskframes.rdr`'. However, they *must* be declared and initialized in the '`<module>.rdr>`' file as described below. The task uses the default values from the '`.rdr`' file if it cannot locate them in '`taskframes.rdr`'.

  See section "Overview" in *LibEditor Programmer's Reference Manual.*
- The file '`<module>.rdr`' is a libReader-style file and consists of the `name`, `struct`, `editor`, `initial`, and `render` lists. There is also an optional `battlemaster` list.

  See section "Overview" in *LibReader Programmer's Reference Manual.*
- The `name` list specifies the name of the task as it should appear on the editor.

  See section "Name Definition" in *LibEditor Programmer's Reference Manual.*
- The `struct` list defines the memory layout of the fields in the task parameters structure, `<module>_PARAMETERS`. For more information on the data types recognized by LibEditor and alignment requirements, See section "Structure Definition" in *LibEditor Programmer's Reference Manual.*
- The `editor` list specifies the task editables that you can modify. You can change the input task parameters (or, alternatively, keep the defaults) through these editables. Note: The order in which the parameters are specified is the order in which they appear (left to right) on the editor. For more information on the usage, internal representation, and description of the available editables,

  See section "Editor Definition" in *LibEditor Programmer's Reference Manual.*
- Each field in the `struct` list must be initialized in the `initial` list. For specifics on how to initialize these parameters,

  See section "Initialization Rules" in *LibEditor Programmer's Reference Manual.*
- The `render` list specifies the `struct` fields that cause the object to be redrawn when their respective editables are changed.

  See section "Rendering Information" in *LibEditor Programmer's Reference Manual.*
- The `battlemaster` list indicates the `struct` fields that require battlemaster privilege in order for the associated widgets to be sensitive. See section "Battlemaster Information" in *LibEditor Programmer's Reference Manual.*

7.5.2.10 Protocol Conversions

The ‘conv_<file_pref>.rdr’ file is necessary if the public structures of the task are modified between software releases. This file supports the loading of "old" scenarios with "new" software. For more information on conversion instructions and examples,

See section “Overview” in *LibPOUtil Programmer’s Reference Manual.*

7.5.2.11 Task Frames Reader File

- Adding the new task to some task frames allows it to be added from the user interface. The file ‘common/src/ModSAF/taskframes.rdr’ specifies the task frames that you have direct control over in the execution matrix. These task frames are comprised of the unit level tasks that are performed during the normal execution of the mission (*primary* tasks) as well as during reactions or actions on contact.
- Task frames are subdivided into two parts, both of which are *also* task frames: PREPARATORY and ACTUAL.
- The PREPARATORY frame consists of tasks that must be run when a unit is in a transitionary phase or a kind of "limbo" state. For example, when a unit is assigned a task, it executes the PREPARATORY frame while it waits for the On Order command.
- Unless special preparations need to be made for the task typically, the PREPARATORY frame consists of the UMixedHalt task for ground vehicles, UFWAHold for fixed winged aircrafts, and URWAHover for rotary winged aircrafts.
- The ACTUAL frame consists of the tasks that define the behavior. For example, the tasks in the ACTUAL frame are executed after the On Order command.
- The primary task is continuously monitored. After the task ends, the frame is considered done.
- Aside from the primary PREPARATORY and ACTUAL tasks, a list of *reaction* tasks are specified, as well. Reactions often belong in BOTH task frames.
- Reactions are not required to be specified, but UActionOnContact is almost always included along with the DEFAULT_REACTIONS macro which initializes some of the reaction parameters with default values.
- The new task macro has the following format:

```
("<task-frame-name>"  <SM_Lib-primary-prep>  <SM_Lib-primary-actual>
                       (<mask-value-pairs>)   ()
 (<SM_Lib> <task-frame>
     (<parameter>  <type>  <value>)
 ...

 )
)
```

where,

- `<task-frame-name` is the name of the task frame (no more than 31 characters)

- `<SM_Lib-primary-prep>` is the primary `PREPARATORY` task model

- `<SM_Lib-primary-actual>` is the primary `ACTUAL` task model

- `<mask-value-pairs>` selects the types of units that should have the option of executing the frame. Mask-value pairs are macro-defined at the beginning of the `taskframes.rdr` file

- Listed after these mask-value pairs is a similar list (which can be empty) indicating specific types of units which appear in the first list, but are *not* allowed to execute the frame

- `SM_Lib` is the name of the task model you are adding to the frame;

- `<task-frame>` specifies the type of frame the task model belongs to (i.e. `PREPARATORY`, `ACTUAL`, or `BOTH`);

- `<parameter>` is a task parameter. Initialize the fields from `<MODULE>_PARAMETERS` in the public header file ('`lib<module>.h`'). By default, the task first checks for parameters here. If a parameter is not initialized, it checks in '`<module>.rdr`'.

- `<type>` and `<value>` specifies the kind of parameter (allowable types are `FORCE`, `CONSTANT`, `REFERENCE`, and `FUNCTION`). For more information on the initialization rules,

See section "Initialization Rules" in *LibEditor Programmer's Reference Manual.*

- For examples, refer to '`taskframes.rdr`' in the directory '`common/src/ModSAF`'.

7.5.2.12 Standard Params Reader File

- Model parameters or system parameters (`<MODULE_PARAMETRIC_DATA`) are declared in the public header file. Initialize the parameters in the file `standard_params.rdr` in the directory `common/src/ModSAF/entities`. Specify the values in a list that begins with the SAF model, followed by the parameter fieldname-value pairings:

```
(<SM_Lib>    (<parameter1>  <value1>)
             (<parameter2>  <value2>)
             ...
)
```

- The `standard_params.rdr` file generally defines architectural and task subclasses that do not differ between vehicles as macros (`GROUND_STD_PARAMS`, `FWA_STD_PARAMS`, `RWA_STD_PARAMS`, `MISSILE_STD_PARAMS`, `DI_STD_PARAMS`, etc). To define something more specific for a given vehicle, include it in the vehicle's parameters file, `<vehicle-name>_params.rdr`.

7.5.2.13 Building the Task

- Build the new task library in its subdirectory in libsrc (as well as librdrconst, and any other libraries that were touched, in their respective subdirectories) with '`gmake`'. Execute '`gmake`' in src/ModSAF to obtain the executable.

7.5.2.14 Documentation

- Every library is documented, especially its public interfaces. Documentation is provided by '`lib<module>.texinfo`' in the library's directory (automatically created by '`tasktemplate`').
- Include an overview of the new library, its functionalities, public interfaces, examples, etc. in the texinfo file.
- Be sure not to reference *other* libraries that are *dependent* on the new library.

7.5.2.15 Debugging

When developing new tasks the following parser commands are useful:

`vehicle <ID> debug task on`

Messages are printed when tasks are started, stopped, suspended, etc. Note: Typing `0` in place of `<ID>` prints debugging messages for *all* vehicles.

`vehicle <ID> debug taskmgr on`

Messages are printed when task frames are started, stopped, etc. on the specified vehicle.

`vehicle <ID> debug <module> on`

This command enables automated debugging generated by the '`fsm2ch`' utility. Messages are printed about events occurring and state transitions.

Many other useful commands are available at the parser line. Type ? at the ModSAF prompt to see list of commands. Some examples include:

`print vehicles`

This command prints the vehicles ID's of all the vehicles.

`veh <ID> show what`

A command that lists the modules for which more information can be shown. Note: `veh` is short for `vehicle`.

7.5.2.16 Rules

Remember to follow these rules when implementing tasks:

- The `PARAMETERS` and `STATE` data structures must have all `ObjectID` references at the beginning of the structures and the reference counts must be set correctly. Failure to do so results in errors when loading saved scenarios containing these tasks, as well as problems with vehicle migration between SAFsims.
- *Never* hold pointers to `PO_DB_ENTRY`s in private `VARS` structures. These pointers become garbage if the corresponding entry is deleted. Instead, hold on to the `ObjectID` of these objects; or keep all object IDs in the public `STATE` structure.
- Write a very good `status` function (see section "statmon_register" in *LibStatMon Programmer's Guide*), as this helps both end users and developers understand the status of the task's behavior.

7.6 Programmer Toolkit

This section describes *padding* and *alignment conventions*. The developer should be familiar with these rules when programming in ModSAF. In addition, the style guide should be read so as to help maintain uniformity in the code.

See section "Files" in *Style Guide*.

7.6.1 Padding

All structures destined for the net should be "Network Friendly". These structures include system parameters (`PARAMETRIC_DATA`), task parameters,(`PARAMETERS`), and shared task state (`STATE`).

For a structure to be "Network Friendly":

- It must have no machine-dependent information. Two prime examples are pointers (such as the libreader symbol) and hash keys (such as the int32 vehicle_id).
- It must adhere to the standard conventions for alignment by explicitly and consistently declaring *padding* to guarantee portability.

7.6.2 Alignment Conventions

Many computers require that data elements be aligned in memory on a multiple of their basic byte or word size. The result is more efficient access to data over the bus. This requirement conflicts with the concept that data array elements be stored in memory contiguously. For example, if a 3-byte array element is loading into address 100 on a machine with a 4-byte word, the next element needs to be stored in address 103 to be contiguous. However, 104 is the next 4-byte aligned address.

To enforce alignment requirements, some C compilers *pad* structures. Padding consists of filling in bytes between struct fields in order to align the fields correctly. As a result, the size of the structure in memory is increased by the number of bytes added as padding.

Unfortunately, due to the use of non-standardized padding methods among different compilers, hardware portability can be problematic. For instance, if a process running on machine A sends a padded structure over the net to a process on machine B, then machine B's compiler supports a different padding mechanism than machine A. Machine B's process assumes a completely different field alignment, thus yielding unpredictable results.

To alleviate this problem, ModSAF data structures that are destined for the net are *explicitly* padded to prevent incompatible compile-time padding.

7.6.2.1 ModSAF Padding Rules

1. The size of a structure is a multiple of its alignment modulus.
2. The byte offset of a field, within a record, is a multiple of the field's alignment modulus.
3. Bit fields generally require 4-byte alignment.

7.6.2.2 What does it mean?

Every data type has an *alignment modulus*. In primitive data types, the alignment modulus is equal to the data type's size. The ModSAF primitive types along with their alignment moduli are:

`char`	1
`int8`	1
`uint8`	1
`int16`	2
`uint16`	2

`int32`	4
`uint32`	4
`float32`	4
`float64`	8

The alignment modulus of a non-primitive data type is equal to the alignment modulus of its most *restrictive* (having a larger alignment attribute) field.

For example, if a structure's largest field (or most restrictive field) is an `int32`, its alignment modulus is 4 bytes. Likewise, the structure's alignment modulus is 2 bytes if its largest field is an `int16`. An array is another example of a non-primitive data type. Hence, its alignment modulus is also equal to the alignment modulus of its most restrictive field. For example the alignment modulus of a `char[2]` array is only 1 byte, not two. The *byte offset* of a structure field is the number of bytes in memory between that field and the top of the structure.

7.6.2.3 Examples

The following are some examples of structures that are properly padded:

1.

```
struct
{
    int8        var1;
    int8        padding;
    int16       var2;
}name;
```

2.

```
struct
{
    int32       var1;
    int16       var2;
    int16       padding;
}name;
```

3.

```
struct
{
    int8        var1;
    int8        var2;
    int16       padding;
    float32     var3;
}name;
```

4.

```
struct
{
    int8        var1;
    int8        pad1[3];
    struct
    {
        int16           x;
        int16           pad2;
        int32           y;
    }name2;
}name1;
```

8 Memory and Processing Time

The performance of ModSAF varies significantly with the following factors:

Hardware platforms

The processor speed, available memory, and networking capabilities of the hardware platform greatly impact the ability of the ModSAF system to simulate large numbers of vehicles.

Compilers The quality of the optimizing compilers available on different platforms vary significantly. In the past, advances in compiler technology have led to more than a doubling of SAF simulation capability.

Networking

There are many options for networking in the DIS environment. In general, attributes of new DIS protocol standards (coordinate system, the broadcast UDP/IP transport, etc.) require more computational overhead than SIMNET protocols.

Accuracy A nominal benchmark of 90% of complete scheduler loops completing within 500 ms over the course of one minute is used as the standard of *acceptable* performance. In some cases, faster tick rates are required; in other cases, this level of performance is overkill. The suitability of a ModSAF system to a particular task requires case-by-case analysis.

Activity There are many measures of activity that impact ModSAF performance, including: the number of vehicles on the DIS battlefield; the density of vehicles in the neighborhood of locally simulated ModSAF entities; and the actions being performed by the locally simulated ModSAF entities.

Taking into account these variables, the ability to rate the performance of ModSAF is an extremely difficult task. The following is used as a worst-case test:

800 Remote Vehicles

The ModSAF '`blaster`' program creates hundreds of vehicles that wander around the database with typical packet rates.

Invincible Vehicles

Since destroyed vehicles require much less computational power than active vehicles, the damage tables (in the file '`dfdam_INVINCIBLE.rdr`') are modified to prevent damage during benchmark exercises.

Unlimited Munitions

Munitions are set to unlimited levels (by including the token `unlimited` in the `SM_Supplies` parameters). This ensures that vehicles do not run out of munitions during the benchmark exercise.

One Vehicle Type

A single vehicle type, the canonical US_M1, is used for all ground benchmarking.

Force on Force Battle

Equal numbers of blue and red M1 vehicles engage in a head-to-head battle. This maximizes the demands of visibility, collision avoidance, and targeting algorithms.

Benchmarking tests are typically run using the most optimal configuration available. This includes:

- The SAFsim process runs on a separate hardware platform from the SAFstation.
- Software is compiled with maximum optimization.

After benchmarking the simulation capabilities, information about processing usage is found using the following tools:

Network Utilization

The levels of packet traffic encountered are monitored using the ModSAF parser interface (`print packets`). Data about numbers of packets sent and received is collected and sorted by protocol family, kind, and size.

Memory Utilization

Memory usage is difficult to accurately measure. The best tools available are operating system utilities, such as '`vmstat`'. However, they often result in incorrect or misleading information.

Processor Utilization

The time spent in different parts of the ModSAF program are measured using profiling tools. These tools are included with most compiler development environments. The best available tool is the '`pixie`' program available on SGI and Mips platforms. The exact number of machine cycles spent in different algorithms can be measured (to the basic-block level) using '`pixie`' and '`prof`'.